

RICE UNIVERSITY  
**Approximate logic circuits: Theory and  
applications**

by

**Mihir Choudhury**

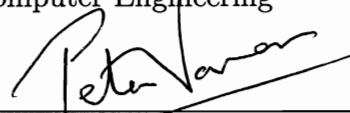
A THESIS SUBMITTED  
IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE  
**Doctor of Philosophy**

APPROVED, THESIS COMMITTEE:



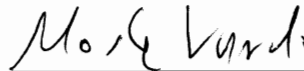
---

Kartik Mohanram, Chair  
Assistant Professor of Electrical and  
Computer Engineering



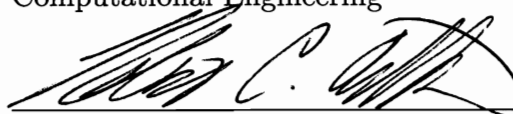
---

Peter J. Varman  
Professor of Electrical and Computer  
Engineering



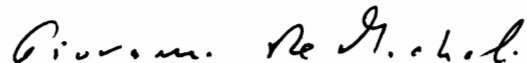
---

Moshe Y. Vardi  
Karen Ostrum George Professor of  
Computational Engineering



---

Robert C. Aitken  
Fellow, ARM Inc., San Jose



---

Giovanni De Micheli  
Professor of Electrical Engineering,  
EPFL, Switzerland

Houston, Texas

May, 2011

## ABSTRACT

Approximate logic circuits: Theory and applications

by

Mihir Choudhury

CMOS technology scaling, the process of shrinking transistor dimensions based on Moore's law, has been the thrust behind increasingly powerful integrated circuits for over half a century. As dimensions are scaled to few tens of nanometers, process and environmental variations can significantly alter transistor characteristics, thus degrading reliability and reducing performance gains in CMOS designs with technology scaling. Although design solutions proposed in recent years to improve reliability of CMOS designs are power-efficient, the performance penalty associated with these solutions further reduces performance gains with technology scaling, and hence these solutions are not well-suited for high-performance designs.

This thesis proposes approximate logic circuits as a new logic synthesis paradigm for reliable, high-performance computing systems. Given a specification, an approximate logic circuit is functionally equivalent to the given specification for a "significant" portion of the input space, but has a smaller delay and power as compared to a circuit implementation of the original specification. This contributions of this thesis include (i) a general theory of approximation and efficient algorithms for automated synthesis of approximations for unrestricted random logic circuits, (ii) logic design solutions based on approximate circuits to improve reliability of designs with negligible performance penalty, and (iii) efficient decomposition algorithms based on approxi-

mate circuits to improve performance of designs during logic synthesis. This thesis concludes with other potential applications of approximate circuits and identifies open problems in logic decomposition and approximate circuit synthesis.

## Acknowledgments

This work would not have been possible without the guidance and support of Prof. Kartik Mohanram. I have known Kartik for six years as my M.S. and Ph.D. thesis advisor. In these years, I have acquired great admiration for Kartik's dedication as a teacher, commitment as an advisor, and perseverance as a researcher. As a graduate student with Kartik, I have received immense freedom to pursue research topics of my interest and I thank Kartik for his patience and trust in my judgment and abilities. I am also grateful to Kartik for giving me the opportunity to attend conferences, workshops, and summer schools around the world that have been a constant source of knowledge and motivation for me. Kartik has been more than an advisor to me. He has been an excellent mentor, from whom I have always felt comfortable seeking advice on both professional and personal matters. There were many moments as a graduate student that I will cherish including the open conversations after paper submissions and conference talks, sharing memories of IIT Bombay, group hikes on Mt. Whitney and Mt. Shasta, and exploring conference towns on foot.

I thank my M.S. thesis committee members, Prof. Peter Varman and Prof. Don Johnson, for expressing interest in my work and providing valuable words of encouragement early in my research career. Their encouragement has always been a source of inspiration for me. I would also like to thank members of my Ph.D. thesis committee — Prof. Peter Varman, Prof. Moshe Vardi, Dr. Robert Aitken, and Prof. Giovanni De Micheli — for taking time out of their busy schedules to provide invaluable feedback on my dissertation.

I am thankful to Dr. Vikas Chandra and Dr. Robert Aitken for giving me the opportunity to spend two summers with the R&D team at ARM. The brain-storming sessions and technical discussions with the R&D team were a unique learning experience for me. I would like to extend a special thanks to Dr. Vikas Chandra for his mentorship and collaboration during my summer internships.

As a graduate student, I have been fortunate to have made some truly amazing

friends. I have enjoyed working and collaborating with my fellow lab members — Quming, Xuebei, Andy, and Masoud. I would like to extend special gratitude to Ajit for his mentorship during my early days as a graduate student. I will always remember the endeavors with Ajit, Meghana, Murari, BD, Kaka Ravi, Rajnish, Ankit, and Ramdas in exploring life outside of Rice university. I am also grateful to the scores of talented friends, with whom I have enjoyed playing table tennis, badminton, and cricket. A special shout out to the Rice Men's Volleyball Club and Rice Squash Club, for sharing moments of glory (and defeat) during tournaments.

I am indebted to my family for their patience and support throughout my education. My father has been a source of my knowledge and a great role model. I am grateful to my mother who, as a teacher, has been instrumental in shaping my academic career. My sister has always encouraged me and believed in my abilities. I acknowledge the support of Rice University, NSF, ARM, Fujitsu, TI, and Intel.

# Contents

Abstract	ii
List of Illustrations	ix
List of Tables	xii
<b>1 Introduction</b>	<b>1</b>
1.1 CMOS technology scaling trends . . . . .	1
1.2 Design challenges . . . . .	2
1.2.1 Reliability challenge . . . . .	4
1.2.2 Performance challenge . . . . .	6
1.3 Contributions of this thesis . . . . .	7
<b>2 Approximate logic functions</b>	<b>10</b>
2.1 Definition of a general approximation . . . . .	11
2.2 Unidirectional approximation . . . . .	13
2.3 Bidirectional approximation . . . . .	15
<b>3 Synthesis of approximate logic circuits</b>	<b>23</b>
3.1 Existing synthesis algorithms and their limitations . . . . .	24
3.2 Motivation for proposed synthesis algorithm . . . . .	28
3.3 Proposed synthesis algorithm . . . . .	31
3.4 Unidirectional approximation . . . . .	33
3.4.1 Type assignment . . . . .	33
3.4.2 Cube selection . . . . .	34
3.5 Bidirectional approximation . . . . .	40

<b>4</b>	<b>Improving reliability with approximate circuits</b>	<b>44</b>
4.1	Design solution for logical errors . . . . .	45
4.1.1	Concurrent error detection . . . . .	46
4.1.2	Concurrent error masking . . . . .	55
4.2	Design solution for timing errors . . . . .	61
4.2.1	Related work: Timing error detection and prediction . . . . .	61
4.2.2	Timing error masking using approximate circuits . . . . .	62
4.3	Example: 2-bit comparator . . . . .	73
<b>5</b>	<b>Time borrowing and error relaying (TIMBER)</b>	<b>75</b>
5.1	Related work and its limitations . . . . .	76
5.2	TIMBER: Overview . . . . .	79
5.3	TIMBER: Circuit design . . . . .	85
5.3.1	TIMBER flip-flop . . . . .	86
5.3.2	Dedicated TIMBER flip-flop . . . . .	91
5.3.3	TIMBER latch . . . . .	92
5.4	Hardware validation . . . . .	98
5.5	TIMBER case study . . . . .	107
<b>6</b>	<b>Performance optimization with approximate circuits</b>	<b>110</b>
6.1	Related work and its limitations . . . . .	111
6.2	Lookahead logic circuits . . . . .	114
6.2.1	Prefix problem . . . . .	115
6.2.2	Extracting timing-critical computation . . . . .	120
6.2.3	Lookahead decomposition . . . . .	122
6.3	Synthesis of lookahead logic circuits . . . . .	125
6.3.1	Primary simplification of $\mathcal{T}$ . . . . .	130

6.3.2	Secondary simplification of $\mathcal{T}$ . . . . .	133
6.3.3	Implication-based simplifications . . . . .	134
6.3.4	Fast adders . . . . .	137
6.4	Results . . . . .	141
6.4.1	Case study: $n$ -bit adder . . . . .	142
6.4.2	Benchmark circuits . . . . .	146
6.4.3	Trend in SPCF . . . . .	148
<b>7</b>	<b>Bi-decomposition of large Boolean functions</b>	<b>150</b>
7.1	Bi-decomposition overview and related work . . . . .	151
7.2	Variable partition using blocking edge graphs (BEGs) . . . . .	153
7.2.1	Blocking condition . . . . .	154
7.2.2	Constructing BEGs . . . . .	157
7.2.3	Variable partition . . . . .	160
7.3	Function decomposition . . . . .	164
7.3.1	Bi-decomposable functions . . . . .	164
7.3.2	Non bi-decomposable functions . . . . .	167
7.4	Bi-decomposition results . . . . .	168
<b>8</b>	<b>Conclusions and future research</b>	<b>177</b>



## Illustrations

2.1	Implication-based unidirectional approximation . . . . .	15
2.2	Implication-based bidirectional approximation. . . . .	17
2.3	Implicit inter-dependent don't cares in a predictor-indicator approximation. . . . .	18
2.4	Predictor-indicator bidirectional approximation. . . . .	19
2.5	Majority bidirectional approximation. . . . .	20
3.1	Algorithm for synthesis of a unidirectional approximation. . . . .	39
4.1	CED based on approximate logic circuits . . . . .	47
4.2	Totally self-checking checker design . . . . .	49
4.3	Concurrent error masking based on approximate logic circuits. . . . .	58
4.4	Timing error masking based on bidirectional approximate circuits. . . . .	67
4.5	(a) 2-bit comparator with two speed-paths, (b) approximate circuit for masking logical errors, and (c) approximate circuit for masking timing errors. . . . .	73
5.1	Critical path distribution between flip-flops. . . . .	80
5.2	TIMBER-based error detection and error masking. . . . .	83
5.3	TIMBER flip-flop (a) design and (b) clock control. . . . .	88
5.4	TIMBER flip-flop error relay logic. . . . .	90
5.5	Two-stage timing error in a TIMBER flip-flop design. . . . .	91

5.6	Dedicated TIMBER flip-flop (a) design and (b) clock control. . . . .	93
5.7	Timing diagram for dedicated TIMBER flip-flop. . . . .	94
5.8	TIMBER latch (a) design and (b) clock control. . . . .	95
5.9	Two-stage timing error in a TIMBER latch design. . . . .	96
5.10	(i) TIMBER flip-flop error relay logic: (a) area overhead and (b) timing slack, (ii) Power overhead for TIMBER flip-flop: (a) without TB and (b) with TB interval, and (iii) Power overhead for TIMBER latch: (a) without TB and (b) with TB interval. . . . .	97
5.11	Hardware setup for measuring error rate with and without TIMBER. . . . .	98
5.12	FPGA implementation of TIMBER flip-flop. . . . .	101
5.13	FPGA implementation of TIMBER latch. . . . .	102
5.14	Timing diagram for timing error rate measurement. . . . .	104
5.15	Error rate vs frequency in linear scale and log scale for TIMBER flip-flop. . . . .	106
5.16	Error rate vs frequency in linear scale and log scale for TIMBER latch. . . . .	107
6.1	Lookahead decomposition using timing-critical computation for general multi-level logic circuits. . . . .	114
6.2	Illustration of (a) cube weight computation and (b) cube selection used in primary and secondary simplification of the technology-independent network $\mathcal{T}$ during the synthesis of lookahead logic circuits. . . . .	124
6.3	Decomposition of $c_{out}$ and $s_3$ of a 4-bit adder in an optimal tree-structured CLA and using lookahead logic circuits. . . . .	141
6.4	In each figure, the left y-axis indicates the levels of logic and the right y-axis indicates the size of the SPCF as a fraction of the input space. Each circuit exhibits an increasing trend in the size of the SPCF when levels of logic are reduced using lookahead logic circuits. . . . .	148

7.1	Obtaining an <b>and</b> bi-decomposition from non-blocking squares. . . . .	156
7.2	Non-blocking squares for <b>and</b> , <b>or</b> , and <b>xor</b> bi-decompositions . . . . .	158
7.3	(a) Incorrect disjoint decomposition indicated by BEG and (b) function with a complete BEG. . . . .	167
7.4	Bi-decomposition using BEGs of (i) $f(a, b, c, d)$ , (ii) $h(b, c, d)$ , and (iii) $h'(b, c, d)$ . (iv) summarizes the bi-decomposition. . . . .	168

# Tables

4.1	Approximation percentage and CED coverage for output cones extracted from benchmark circuits. . . . .	51
4.2	Area-power overhead and CED coverage for MCNC benchmark circuits.	52
4.3	Technology-independence of CED coverage . . . . .	54
4.4	Area and power overhead for concurrent masking of logical errors. . .	59
4.5	Accuracy vs. runtime for computing speed-path characteristic function with different approaches . . . . .	63
4.6	Area and power overhead for 100% concurrent masking of timing errors on speed-paths. . . . .	70
5.1	Comparison of various techniques for online timing error resilience. .	79
6.1	Simplification rules for the generalized Shannon decomposition $f = gf_g + \bar{g}f_{\bar{g}}$ . . . . .	129
6.2	Comparison of best AIG levels after timing optimization of an $n$ -bit adder, $n = 2, 4, 8, 16, 32$ . . . . .	143
6.3	Comparison of the proposed technique with the best algorithms in SIS, ABC, and the industry-standard synthesizer . . . . .	145
7.1	Comparison of the proposed technique with the best algorithms in FBDD, SIS, ABC, and the industrial tool <sup>†</sup> . . . . .	171

7.2	Comparison of the proposed technique with the best algorithms in FBDD, SIS, ABC, and the industrial tool . . . . .	172
-----	---	-----

# Chapter 1

## Introduction

### 1.1 CMOS technology scaling trends

For over half a century, faster transistors and higher device integration offered by CMOS technology scaling has driven performance gains in computing systems. However, today, computing in the information age demands higher processing power, but sustaining performance gains with CMOS technology scaling is becoming more challenging as transistor dimensions are scaled to few tens of nanometers.

Variability in CMOS designs is a major hurdle for technology scaling today. Variability refers to the deviation in transistor characteristics, observed during the normal operation of a chip, relative to the nominal characteristics determined during design. Manufacturing *process variations*, e.g., channel length variations, random dopant fluctuations, and oxide thickness variations cause variations in transistor characteristics such as threshold voltage and  $I_{\text{on}}/I_{\text{off}}$  after fabrication. For instance, for devices in 65 nm CMOS process technology, the  $3\sigma$  variations in threshold voltage was observed to be 30% of the nominal value [1], significantly higher than prior CMOS technologies. The variations in transistor characteristics cause each fabricated chip to have unique noise robustness, critical path delay, and power dissipation.

Unlike manufacturing process variations, *dynamic variations* can cause transistor characteristics to vary during normal operation of a chip. Single-event upsets due to radiation strikes causes charge deposition resulting in unwanted glitches in the drive current of devices. Temperature/supply voltage variations and crosstalk cause delay variations in logic gates and interconnects. Furthermore, in recent CMOS process technologies, aging and wearout related effects such as negative bias temperature instability (NBTI) and positive bias temperature instability (PBTI) have been observed. These effects increase the threshold voltage of transistors over long periods of time, e.g., the threshold voltage of devices in the 65nm process technology were observed to increase by 38% over a period of 10 years [1]. With the introduction of high- $k$  metal dielectrics in the 45nm CMOS process, devices are more susceptible new aging mechanisms such as time dependent dielectric breakdown (TDDB) [2].

## 1.2 Design challenges

The traditional design strategy to account for the impact of variability was to add worst-case margins, such as timing margins for delay variations, during design. As variability increased with technology scaling, the hardware overhead for adding worst-case margins during design also increased. Consequently, design-for-manufacturability (DFM) techniques such as optical proximity correction (OPC), resolution enhancement techniques (RET) and design-for-yield (DFY) techniques such as statistical timing [3], statistical design optimization techniques [1; 4] were introduced to re-

duce the hardware overhead for maintaining high manufacturing yield at the target specification. As variability continues to increase and new variability mechanisms are discovered with technology scaling, DFM and DFY techniques will also be inadequate for ensuring reliable operation during normal operation of a chip in future CMOS technologies. Hence, researchers today are exploring new design solutions that can detect and recover from errors arising due to variability during normal operation of a chip. Although traditional approaches for designing reliable systems, e.g., duplication, triple modular redundancy (TMR) [5], and output encoding [6; 7] can guarantee high levels of reliability, the associated performance and power overhead prevents these approaches from being incorporated into mainstream applications. To reduce performance and power overhead, researchers today acknowledge that a hierarchical approach, based on cost-effective design solutions for improving reliability at different levels of design abstraction, is more suitable for designing reliable systems for mainstream applications. The need for cost-effective reliable design solutions at the application, architecture, logic, and circuit level has triggered a paradigm shift in CMOS design, from the conventional performance-power trade-off to reliability-performance-power trade-off. The remainder of this section describes the challenges posed by variability in future CMOS technologies and the limitations of existing design solutions.



### 1.2.1 Reliability challenge

Manufacturing defects such as gate-source shorts and oxide traps can manifest as *intermittent gate failures* during normal operation of a chip. Such failures are increasing during normal operation of a chip because extensive post-manufacturing test and burn-in, used to identify faulty chips, are becoming less effective in advanced CMOS technologies due to the increased chance of thermal runaway [8]. Single-event upsets, known to cause bit flips in semiconductor memories for many years, are also beginning to cause *transient gate failures* in combinational logic. Both intermittent and transient failures, if propagated through a logic circuit, result in *logical errors* at the output of the logic circuits. Logical errors can corrupt the state of a system, and since these errors are not timing-related, they cannot be eliminated by reducing frequency of operation.

Although logical errors are predicted to increase in future CMOS technologies, *timing errors* arising due to gate delay variations on critical paths are the dominant concern in current CMOS technologies. Delay variations can be caused due to dynamic variability effects such as temperature variations, supply voltage fluctuations, and clock jitter. Transistor aging effects, caused due to negative bias temperature instability (NBTI), positive bias temperature instability (PBTI), and time-dependent dielectric breakdown (TDDB), can cause gradual slowdown of transistor switching speed over time. Transistor aging effects, observed to increase in sub-45nm CMOS technologies, also pose a dilemma during post-manufacturing test and burn-in — on

one hand, extensive post-manufacturing test and burn-in must be performed to detect aging effects and on the other hand, extensive post-manufacturing test and burn-in can degrade good quality, robust chips by speeding-up the aging process in these chips.

Due to the increasing reliability concern, in recent years, there has been great interest in exploring cost-effective design solutions for improving reliability. Exploring circuit and logic design solutions are particularly appealing because (i) impact of variability on reliability can be assessed accurately at the circuit and logic levels of design abstraction and (ii) cost-effective design solutions for improving reliability are possible due to the fine-grained control over the use of hardware resources during circuit and logic design. Examples of recent circuit and logic design solutions include partial error detection/masking [9; 10; 11; 12] for improving reliability of combinational logic circuits to intermittent and transient failures, robust flip-flop designs to prevent data corruption due to single-event upsets [13; 14], techniques for detecting/predicting timing errors using sensors and transition detectors [15; 16; 13; 17], and using double sampling of the data signal [18; 19; 20; 21]. Although the aforementioned circuit and logic design solutions for improving reliability are power-efficient, these techniques hurt the performance of a design and hence, are not well-suited for improving reliability in high-performance designs. The performance penalties associated with these techniques are highlighted below.

### 1.2.2 Performance challenge

Variability also reduces the performance of designs. Process variations have been observed to cause a frequency spread of 30% in the 180nm CMOS technology [22]. The spread in frequency of operation is significantly higher in the latest 22nm CMOS technology. Since the frequency of operation is determined by the slowest critical path in a chip, process variations hurts performance because a bulk of the fabricated chips operate at a frequency lower than the nominal frequency. Furthermore, existing design solutions for improving reliability through error detection, masking, and prediction also impose a performance penalty on the original design.

Error detection techniques [15; 16; 10; 13] have to rely on roll-back or local instruction relay to correct errors. Roll-backs incur significant performance penalty and instruction replays require extensive hardware support, especially for high-performance designs that typically have deep pipelines and complex control logic.

Error masking techniques [10; 11; 12] eliminate the need for a roll-back or instruction replay by adding extra hardware for in-line error correction dynamically during runtime. For instance, techniques such resynthesis [11] and rewiring [12] improve reliability by restructuring logic to increase logical masking within the combinational logic circuit. However, often the restructured logic circuit has a larger critical path delay and hence, hurts performance. A technique based on triplicating the most critical portions of a logic circuit [10] is power-efficient and has a low performance overhead, but is susceptible to common mode failures [23].

Error prediction techniques [17; 20] also eliminate the need for a roll-back or instruction replay by detecting slowdown of critical paths before the occurrence of a timing error. However, these techniques can only predict timing errors arising due to gradual slowdown of critical paths, e.g., due to NBTI, PBTI, and TDDB, but not due to fast-changing dynamic variability effects such as temperature/supply voltage variations and clock jitter. Furthermore, error prediction techniques also incur a performance penalty due to the timing guard-band required for detecting slowdown of critical paths.

To summarize, increasing variability is reducing performance gains offered by CMOS technology scaling and degrading reliability of CMOS designs. Existing design solutions for improving reliability of high-performance designs incur a performance penalty, and hence further reduce performance gains offered by CMOS technology scaling. The battle between reliability and performance during design is evident from the recent recall of the Cougar Point 6 series chip-sets for the Intel's 32nm Sandy Bridge microprocessor. The recall, estimated to cost \$1 billion, was initiated after Intel found that the performance of the SATA ports in a fraction of the chips sold to consumers would degrade rapidly over time and eventually fail.

### **1.3 Contributions of this thesis**

This thesis proposes approximate logic circuits as a new logic design paradigm for designing reliable, high-performance computing systems. Given a specification, an

approximate logic circuit can predict with certainty the value of the given specification for a specified portion of the input space, but has a smaller delay, area, and power as compared to a circuit implementation of the original specification. Based on a general theory of approximation for Boolean functions and efficient synthesis algorithms developed in this thesis, approximate logic circuits can be automatically generated for any given logic circuit (specification). This thesis demonstrates that approximate logic circuits can be used to improve both reliability and performance of combinational logic circuits.

- Improving reliability: This thesis demonstrates that approximate logic circuits can be used to improve reliability by detecting/masking logical and timing errors at during normal operation in a more power-efficient than existing error detection/masking approaches. To further reduce power overhead for timing error masking using approximate circuits, this thesis proposes TIMBER, an architecture for masking timing errors by borrowing time from successive pipeline stages. Unlike existing approaches, error masking using both approximate circuits and TIMBER incurs negligible performance penalty, and hence is well-suited for improving reliability of high-performance designs.
- Improving performance: Existing tools optimize performance during logic synthesis using fast structural logic transformations, instead of time consuming functional decomposition algorithms, to ensure scalability to large designs. Consequently, these tools often deliver designs with sub-optimal performance. This

thesis demonstrates that approximate circuits provide an efficient means for exploring functional decompositions because, unlike existing functional decomposition algorithms, they leverage the decomposition structure of the given logic circuit. Using function decomposition insights from approximate circuits, this thesis also proposes a variable partition algorithm for function decomposition that, unlike existing variable partition algorithms, provides optimal size variable partitions and is scalable to industrial size benchmark circuits. On average, the proposed algorithms can improve performance by 10–25% over state-of-the-art logic synthesis tools, with comparable computational cost.

This thesis is organized as follows. Chapter 2 proposes a general theory of approximation for Boolean functions and Chapter 3 proposes algorithms for synthesizing logic circuits for approximations. Chapter 4 and Chapter 5 demonstrate application of approximate circuits for improving reliability of logic circuits. Chapter 6 and Chapter 7 demonstrate application of approximate circuits to improve performance during logic synthesis. Chapter 8 concludes the thesis by highlighting open problems and directions for future research.

## Chapter 2

### Approximate logic functions

Given a Boolean specification, an approximation can predict with certainty the value of the given Boolean function for a specified portion of the input space. Approximation of a given Boolean specification can capture useful circuit functionality in a logic circuit that has a smaller hardware cost (delay, area, and power) as compared to a logic circuit for the given specification.

Several instances of simple approximations have been proposed in literature to improve area, delay, testability, and dynamic power consumption of logic circuits. However, due to the lack of a formal theory, approximations proposed in literature were limited to either simple approximations or applied only to regular logic circuits. This thesis is the first to propose a formal theory for general approximation of a Boolean function. This general theory of approximation subsumes instances of approximations used in literature, and has enabled us to develop algorithms for automatically generating approximations for unrestricted random logic circuits. The remainder of this chapter proposes a general definition of an approximation and illustrates how instances of approximations in existing literature are special cases of the general approximation.

## 2.1 Definition of a general approximation

A Boolean function with  $n$  inputs is a function of the form  $f : B^n \rightarrow B$ , where  $B = \{0, 1\}$ . A Boolean specification with  $n$  inputs is a function  $g : B^n \rightarrow \{0, 1, -\}$ , where “ $-$ ” stands for a don’t care, i.e., the value of  $g$  is not specified for input combinations mapped to  $-$ . A specification  $g$  is said to be complete if no input combination of  $g$  is mapped to  $-$ , and incomplete otherwise. Thus, a Boolean function is a completely specified Boolean specification. A Boolean specification  $g$  can also be described as a partition of input space  $B^n = \{0, 1\}^n$  into three sets: (i) on-set: set of inputs in  $B^n$  where  $g$  is a 1, (ii) off-set: set of inputs in  $B^n$  where  $g$  is a 0, and (iii) don’t-care-set: set of inputs in  $B^n$  where  $g$  is a  $-$ . These three sets can be identified by three characteristic Boolean functions: (i) on-set characteristic function,  $g_{\text{on}}$ , is 1 for inputs in the on-set of  $g$ , 0 otherwise, (ii) off-set characteristic function,  $g_{\text{off}}$ , is 1 for inputs in the off-set of  $g$ , 0 otherwise, and (iii) don’t-care-set characteristic function,  $g_{\text{dc}}$ , is 1 for inputs in the don’t-care set of  $g$ , 0 otherwise. Since the on-set, off-set, and don’t-care-set are exhaustive and mutually exclusive sets in  $B^n$ , the pair-wise products of  $g_{\text{on}}$ ,  $g_{\text{off}}$ , and  $g_{\text{dc}}$  are 0, and  $g_{\text{on}} + g_{\text{off}} + g_{\text{dc}}$  is 1.

An approximation of a given Boolean specification,  $g$ , predicts the correct value of the  $g$  for a specified portion of the input space *and* indicates uncertainty about the value of the  $g$  for the rest of the input space. Note that indicating uncertainty about the value of the specification is essential to ensure that the specification is not altered when an approximation is used for concurrent error masking. Thus, given a



characteristic Boolean function  $S : B^n \rightarrow B$  that specifies the portion of the input space to predict for  $g$ , an approximation,  $\hat{g}$ , is defined as a set of two Boolean  $\hat{g} = \{\tilde{g}, e\}$ , where  $\tilde{g}$  and  $e$  having the same inputs as  $g$ . The Boolean functions  $\tilde{g}$  and  $e$  are referred to as the predictor and indicator functions, respectively. The predictor function,  $\tilde{g}$ , predicts the value of  $g$  and the indicator function,  $e$ , indicates uncertainty about the value of  $g$ . In our notation for indicating uncertainty about the value of  $g$ , when  $e$  is 1,  $\tilde{g}$  is equal to  $g$  and when  $e$  is 0  $\tilde{g}$  may or may not be equal to  $g$ . Note that the inputs in the don't care set of an incomplete specification  $g$  are don't cares for both  $\tilde{g}$  and  $e$ . In Boolean algebra, the relation between the predictor and indicator functions of an approximation can be expressed as:

$$g \approx \hat{g} = \{\tilde{g}, e\} \text{ s.t. } ((\tilde{g}g_{\text{off}} + \bar{\tilde{g}}g_{\text{on}}) \Rightarrow \bar{e}) = 1 \quad (2.1)$$

The on-set of the indicator function  $e$ , i.e., when  $e$  is 1, is the input sub-space on which the approximation,  $\hat{g}$ , correctly predicts the given specification  $g$ . This input sub-space is denoted as  $\Sigma$ . Note that the specified input sub-space ( $S$ ) is typically different from the input sub-space ( $\Sigma$ ) for which the approximation predicts the given specification correctly. This is because when an approximation for a specified input sub-space  $S$  is implemented as a logic circuit, by allowing  $\Sigma$  to be different from  $S$ , the hardware overhead (area, delay, and power) for the approximate logic circuit can be reduced. The quality of the approximation (approximation percentage) is evaluated as the fraction of the specified input sub-space  $S$  that is correctly predicted by the

approximate logic circuit and can be computed as  $\frac{|S \cdot \Sigma|}{|S|}$ .

This thesis classifies approximations into two categories — unidirectional and bidirectional. The reason for this classification will become clear when the applications of approximate logic circuits to improve reliability of logic circuits are described in Chapter 4. This chapter formalizes definitions of unidirectional and bidirectional approximations.

## 2.2 Unidirectional approximation

An approximation is called unidirectional if the input sub-space that is predicted correctly ( $\Sigma$ ) is either a subset of the on-set or a sub-set of the off-set of  $g$ . A unidirectional approximation, by definition, is an implication function. Implication functions have been widely used in literature to improve area, testability, and more recently, reliability of logic circuits. Kunz et al. propose recursive learning, a technique based on implications, to identify and eliminate redundancy in logic circuits. Recursive learning uses implication relations within a logic circuit as candidates for eliminating redundancy. The authors show that eliminating redundancy using recursive learning can improve both area and testability of designs [24]. Recently, Krishnaswamy et al. propose a combination of random input vector simulation and SAT algorithms for identifying implications in a logic circuit. In contrast to recursive learning, Krishnaswamy et al. leverage existing redundancy in the form of implication relations to improve robustness of a logic circuit to soft errors at the expense of circuit area.

However, Krishnaswamy et al. do not analyze the impact of their technique on the testability of their circuit [11].

An implication function,  $\tilde{g}$ , of  $g$  satisfies  $\tilde{g} \Rightarrow g$ . When  $\tilde{g}$  is 1,  $\tilde{g}$  is equal to  $g$ , i.e.,  $\tilde{g}$  predicts the correct value of  $g$ . When  $\tilde{g}$  is 0,  $\tilde{g}$  may or may not predict the correct value of  $g$ , i.e.,  $\tilde{g} = 0$  indicates uncertainty about the value of  $g$ . Thus, the unidirectional approximation  $\hat{g} = \{\tilde{g}, \tilde{g}\}$  satisfies the condition for an approximate logic function (Eqn. 2.1). The Boolean function  $\tilde{g}$  is also called an under-approximation or on-set unidirectional approximation of  $g$ . Similarly, an over-approximation or off-set unidirectional approximation of  $g$  satisfies  $\bar{\tilde{g}} \Rightarrow \bar{g}$ . An off-set unidirectional approximation for a Boolean function  $g$  is illustrated using Karnaugh map (K-map) in Figure 2.1. The shaded cells of the K-map of  $g$  indicate the specified input sub-space,  $S$ , of  $g$  and the shaded cells of the K-map of  $\tilde{g}$  indicate the input sub-space,  $\Sigma$ , that is predicted correctly by  $\tilde{g}$ . Since, 6 out of the 7 minterms in  $S$  are predicted correctly by  $\tilde{g}$ , the approximation percentage is 85.7%.

The number of unidirectional approximate logic functions grows exponentially as the size of the specified input sub-space,  $S$ , decreases. If  $\mathcal{N}$  is the on-set of  $g$  and  $S \subset \mathcal{N}$ , then for an on-set unidirectional approximation  $\tilde{g}$  with an approximation percentage of 100%,  $\tilde{g}$  is 1 for all inputs in  $S$ . For the inputs outside of  $S$ , but in  $\mathcal{N}$ ,  $\tilde{g}$  is a don't care, i.e.,  $\tilde{g}$  may or may not predict inputs in  $\mathcal{N} \setminus S$  correctly. Each don't care can be assigned either a 0 or a 1 to get a different approximation. Thus, there are  $2^{|\mathcal{N}| - |S|}$  ways of assigning the don't cares, and hence there are  $2^{|\mathcal{N}| - |S|}$  on-set

unidirectional approximations of  $g$ . Note that the size of  $\mathcal{N}$  and  $S$  are exponential in the number of inputs of  $g$ .

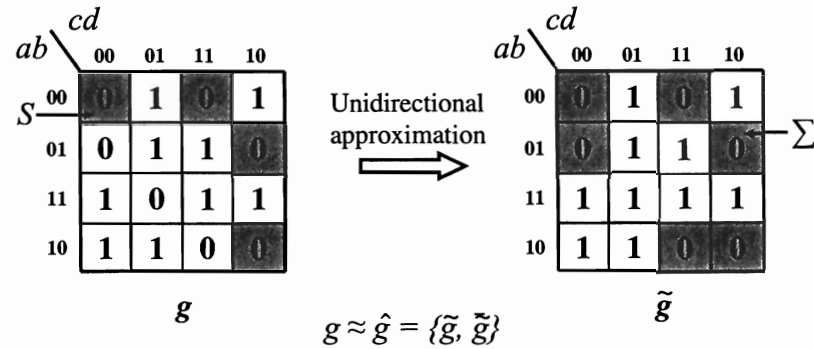


Figure 2.1 : Implication-based unidirectional approximation

## 2.3 Bidirectional approximation

An approximation is called bidirectional the input sub-space that is predicted correctly ( $\Sigma$ ) contains portions of both the on-set and the off-set of  $g$ . Unlike a unidirectional approximation, there are multiple ways of deriving a bidirectional approximation of  $g$ . In this section, three ways of deriving a bidirectional approximation are described: (i) implication-based bidirectional approximation, (ii) predictor-indicator bidirectional approximation, and (iii) majority bidirectional approximation. The implication-based bi-directional approximation, used in existing circuit approximation applications, e.g., [25; 26], cannot be used to derive all bidirectional approximations. On the other hand, the predictor-indicator and majority bidirectional approximations can be used to derive all bidirectional approximations.

**Implication-based bidirectional approximation:** A bidirectional approximation of  $g$  can be obtained using two unidirectional approximations — an on-set unidirectional approximation  $\tilde{g}_1$  and an off-set unidirectional approximation  $\tilde{g}_0$ . Besides exploring implications within a given logic circuit, synthesis of implication functions as stand-alone logic circuits have been proposed for reducing dynamic power consumption. Alidina et al. propose pre-computation architectures to reduce dynamic power consumption in sequential circuits. Pre-computation architectures use Boolean implications to predict outputs of a combinational logic stage. When a correct output is predicted, pre-computation architectures eliminate switching activity on the next clock cycle by disabling the inputs to the combinational logic stage, thus reducing power consumption [25; 27; 28]. Extensions of the pre-computation architecture to further reduce power dissipation [29] have also been proposed. Saldanha et al. reduce the delay of a logic circuit by converting critical paths into false paths using implication-based approximations [26]. Implications have also been used to reduce power consumption for logic blocks, e.g., clock gating and branch-prediction, that do not affect the correctness of computation.

An implication-based bidirectional approximation Since  $\tilde{g}_1 \Rightarrow g$ ,  $\tilde{g}_1 = 1$  predicts the correct value for a portion of the on-set of  $g$ . Similarly, since  $\overline{\tilde{g}_0} \Rightarrow \overline{g}$ ,  $\tilde{g}_0 = 0$  predicts the correct value for a portion of the off-set of  $g$ . Note that  $\tilde{g}_1$  and  $\tilde{g}_0$  have the same value when  $\tilde{g}_1 = 1$  and when  $\tilde{g}_0 = 0$ , i.e., both  $\tilde{g}_1$  and  $\tilde{g}_0$  predict the correct value of  $g$  when  $\tilde{g}_1 = \tilde{g}_0$ . Hence, either  $\tilde{g}_0$  or  $\tilde{g}_1$  can be used as the predictor function

$\tilde{g}$  and  $\tilde{g}_0 \oplus \tilde{g}_1$  is the indicator function  $e$ . Figure 2.2 illustrates an implication-based bidirectional approximation. Note that if either  $u$  is a constant 0 function or if  $v$  is a constant 1 function, the implication-based bidirectional approximation reduces to an implication-based unidirectional approximation.

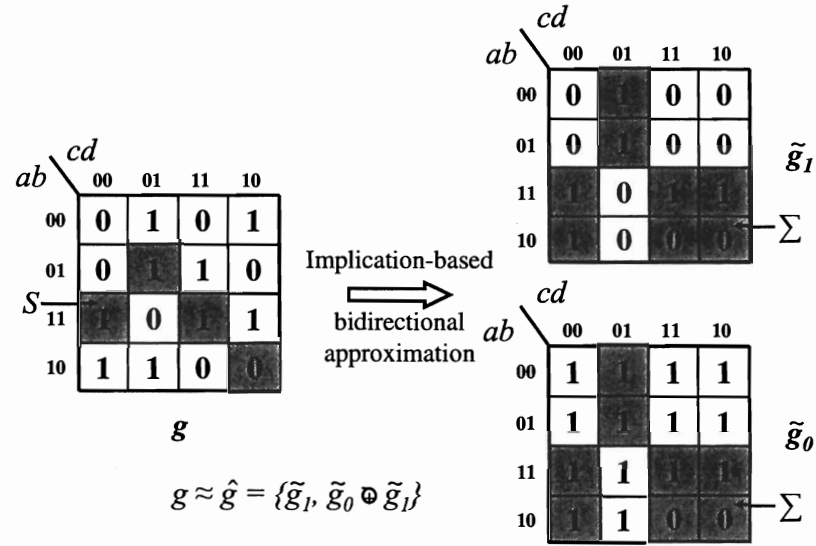


Figure 2.2 : Implication-based bidirectional approximation.

**Predictor-indicator bidirectional approximation:** A predictor-indicator approximation is obtained directly using a predictor function  $\tilde{g}$  and an indicator function  $e$ . In a predictor-indicator approximation, the predictor function  $\tilde{g}$  and the indicator function  $e$  may not have implication relations with the given specification  $g$ , but instead,  $\tilde{g}$  and  $e$  have inter-dependent don't cares. The inter-dependent don't cares between  $\tilde{g}$  and  $e$  is described below.

To achieve a 100% approximation percentage using a predictor-indicator approximation, the predictor function  $\tilde{g}$  must be equal to  $g$  and the indicator function  $e$  must

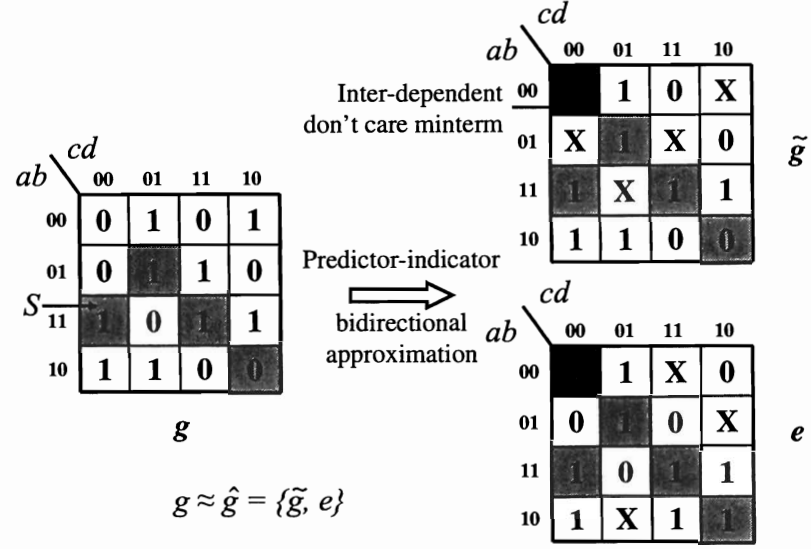


Figure 2.3 : Implicit inter-dependent don't cares in a predictor-indicator approximation.

be 1 for the entire specified input space  $S$ . For the remaining portion of the input space  $\bar{S}$ , the only condition that decides the values of  $\tilde{g}$  and  $e$  is that the indicator function  $e$  must not incorrectly indicate that the  $\tilde{g}$  predicts the correct value of  $g$ . Thus, in  $\bar{S}$ , if  $\tilde{g}$  predicts  $g$  correctly,  $e$  can be either a 0 or a 1, i.e.,  $e$  is a don't care. On the other hand, if  $e$  is a 0, then  $\tilde{g}$  can either be a 0 or a 1, i.e.,  $\tilde{g}$  is a don't care. Thus, in  $\bar{S}$ , the predictor and indicator functions have *implicit inter-dependent don't cares*. The don't cares are implicit because these don't cares are not present in the given specification  $g$ , but arise implicitly from the definition of predictor-indicator bidirectional approximation. The don't cares are inter-dependent because the don't cares in  $\tilde{g}$  depend on the value of  $e$  and vice-versa. There are many combinations of implicit inter-dependent don't cares in  $\tilde{g}$  and  $e$ . Figure 2.3 shows one combination of implicit

inter-dependent don't care in  $\tilde{g}$  and  $e$ . Each of these don't cares in  $f$  and  $e$  can be assigned either a 0 or a 1 to obtain a predictor-indicator approximation as shown in Figure 2.4. Using various combination of implicit inter-dependent don't cares and the assignment of these don't cares to a 0 or a 1, every bidirectional approximation can be expressed as a predictor-indicator approximation and thus, a predictor-indicator approximation is the most general form of bidirectional approximation.

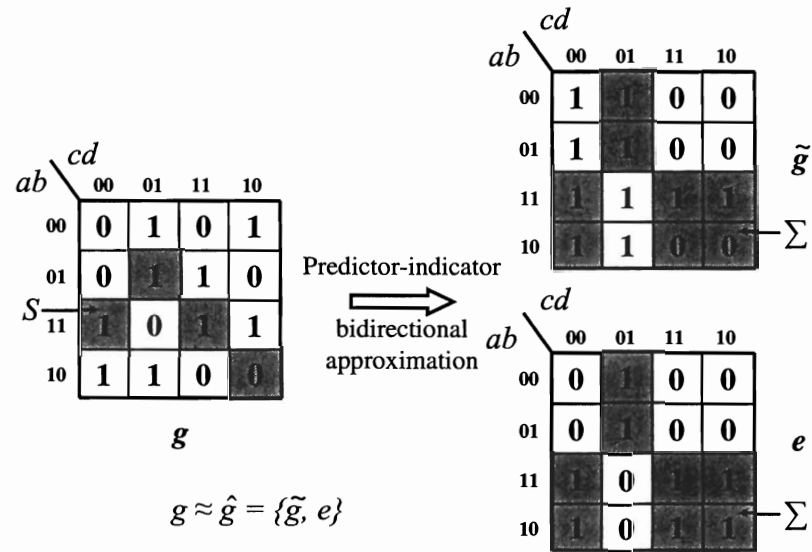


Figure 2.4 : Predictor-indicator bidirectional approximation.

**Majority bidirectional approximation:** A majority approximation is obtained using two functions,  $\tilde{g}_1$  and  $\tilde{g}_2$ , such that if  $\tilde{g}_1 = \tilde{g}_2$  then  $g = \tilde{g}_1 = \tilde{g}_2$ . As with the predictor-indicator approximation,  $\tilde{g}_1$  and  $\tilde{g}_2$  may not have implication relation with  $g$ , but  $\tilde{g}_1$ , but instead,  $\tilde{g}_1$  and  $\tilde{g}_2$  have inter-dependent don't cares.

To achieve a 100% approximation percentage using a majority approximation,  $\tilde{g}_1$  and  $\tilde{g}_2$  must be equal to  $g$  for the entire specified input space  $S$ . For the remaining



portion of the input space  $\bar{S}$ , at least one of  $\tilde{g}_1$  and  $\tilde{g}_2$  must be equal to  $g$ . Hence, for a minterm in  $\bar{S}$ , if  $\tilde{g}_1 = g$ , then  $\tilde{g}_2$  is a don't care and vice-versa.

Figure 2.5 illustrates a majority approximation. The functions  $\tilde{g}_1$  and  $\tilde{g}_2$  can also be obtained from  $g$  by inverting the value of  $g$  for sets of minterms  $S_1$  and  $S_2$ , respectively such that  $S_1 \cap S_2 = \phi$ . Note that a predictor-indicator approximation can be obtained from a majority approximation by setting the predictor function  $\tilde{g} = \tilde{g}_1$  or  $\tilde{g} = \tilde{g}_2$  and the indicator function  $e = \tilde{g}_1 \oplus \tilde{g}_2$ . Similarly, a majority approximation can be obtained from a predictor-indicator approximation by setting either  $\tilde{g}_1 = \tilde{g}$  and  $\tilde{g}_2 = \tilde{g} \oplus e$  or vice-versa.

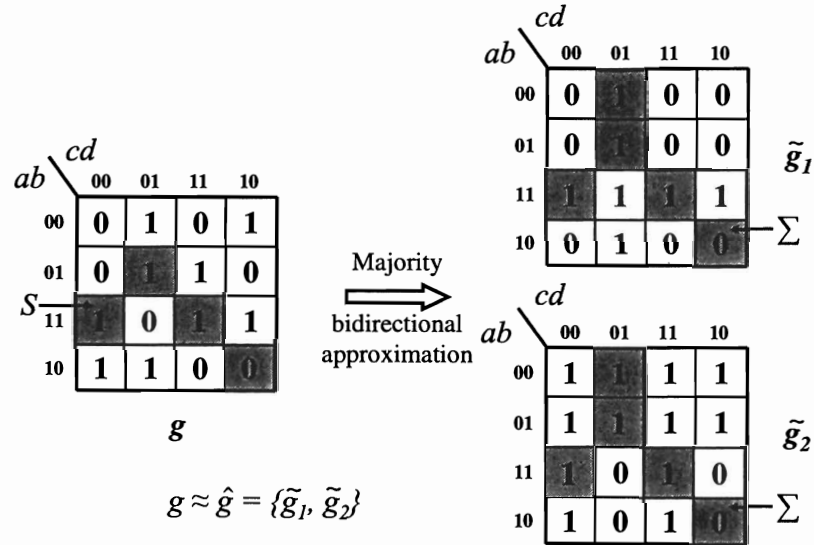


Figure 2.5 : Majority bidirectional approximation.

The number of predictor-indicator (or majority) bidirectional approximations of  $g$  grow exponentially as the size of the specified input space,  $S$ , decreases. This is illustrated for a predictor-indicator bidirectional approximation. In a predictor-

indicator bidirectional approximation, the value of the predictor function  $\tilde{g}$  is fixed for the inputs in  $S$ , i.e.,  $\tilde{g}(S) = g(S)$  and the indicator function,  $e$ , is 1 to indicate that  $\tilde{g}$  predicts  $g$  correctly. For inputs outside of  $S$  (in  $B^n \setminus S$ ),  $\tilde{g}$  and  $e$  have implicit inter-dependent don't cares, i.e., the indicator function is a don't care if  $\tilde{g}$  predicts  $g$  correctly, and 0 otherwise. Thus, if  $\tilde{g}$  predicts  $g$  correctly for  $s$  inputs in  $\overline{S}$ , then there are  $s$  don't cares in the indicator function. There are  $2^s$  ways of assigning these  $s$  don't cares. Hence, the total number of bidirectional approximate functions for  $g$  is given by  $\sum_{s=0}^{|\overline{S}|} \binom{|\overline{S}|}{s} 2^s = 3^{|\overline{S}|}$ . To summarize, the space of bidirectional approximations is rich and exploring this space during synthesis of bidirectional approximations is challenging. In the next section, we will describe efficient algorithms for synthesizing approximate logic circuits.

Without this theory, the application of general approximations was limited to regular logic circuits, e.g. arithmetic circuits. The general approximation for regular logic circuits such as adders is used to improve performance using speculation. By approximating the Boolean specification, speculative techniques improve operating frequency because the logic circuit for the approximation has a smaller delay. However, the logic circuit for the relaxed specification may occasionally compute an incorrect value. When an incorrect value is computed, the error is corrected by a roll-back or local instruction relay. Speculative techniques proposed in literature leverage designer knowledge for simplifying a logic circuit, and hence have only been applied to regular designs such as adders [30; 31], rename and issue logic [30], and not to

irregular multi-level logic circuits because there is no algorithm for automatically synthesizing general approximations.

## Chapter 3

### Synthesis of approximate logic circuits

In the discussion so far, we have defined a general approximation and described two kinds of approximate logic functions — unidirectional and bidirectional. We have also shown that, given a logic specification and a specified input space to approximate, the number of approximate logic functions is exponential in the size of the input subspace that is not targeted by the approximation. Due to the large space of possible approximations, designing efficient algorithms for automated synthesis of approximate circuits is challenging. Given a specification, for approximations to be a useful design solution for improving reliability and performance, synthesis algorithms must be able to identify approximate logic functions and generate approximate logic circuits with the following characteristics.

- **Small size:** The synthesis algorithm must generate an approximate logic circuit that has a smaller delay and power as compared to the given logic circuit. Between delay and power, reducing delay has a higher priority because our ultimate goal is to use the approximate logic circuit in design solutions for reliable, high-performance systems, where a performance penalty is not acceptable.
- **Flexibility:** The synthesis algorithm should also be able to use a specified input

sub-space to guide the synthesis process towards correctly predicting inputs in the specified input sub-space. This flexibility in specifying the input sub-space for approximation will be useful for customizing and targeting approximate circuits towards a specific application, e.g., improving reliability of a logic circuit against logical errors or timing errors.

- **Scalability:** The synthesis algorithm should be computationally efficient so that they can be applied to combinational logic circuits found in modern industrial microprocessors.

This thesis develops synthesis algorithms for approximating a given specification represented as a gate-level netlist. A gate-level netlist is an interconnection of logic gates as a directed acyclic graph (DAG). The logic gates in the netlist must have an associated delay, area, and power model. For instance, the logic gates in the gate-level netlist can be mapped gates from a technology library or even, just simple `and` gates, `or`, `not` gates with a technology-independent delay, area, and power model for each logic gate.

### 3.1 Existing synthesis algorithms and their limitations

Existing techniques for synthesis of approximate logic circuits can be broadly divided into two categories.

- Bottom-up approach is not scalable: In a bottom-up approach, the given gate-level netlist is first completely collapsed to obtain a Boolean function for each

primary output. The synthesis of approximate logic circuit is then a two step process: (i) assigning the don't cares introduced by the approximation to obtain a completely specified approximate function and (ii) synthesizing the completely specified approximate function into an approximate logic circuit. An example of a bottom-up approach is based on minimizing the sum-of-product (SOP) representation of the approximate logic function using a two level minimizer, e.g. ESPRESSO [32]. Multi-level logic optimization can then be used to synthesize the approximate logic circuit from the minimized SOP expression. Although, this approach can leverage the rich space of don't cares introduced by the approximation, it is not scalable to circuits with more than 15–20 primary inputs. If the given specification can be represented as a binary decision diagram (BDD), then BDD operators such as restrict and constrain can be used to obtain an approximate logic function by assigning the don't care space introduced by the approximation. In [25], the BDD for an approximation is converted into a multi-level approximate logic circuit by implementing the BDD as a logic circuit with multiplexers. However, this approach often results in an approximate circuit with large delay and power footprint. Alternatively, the BDD for the approximation can be decomposed using BDD-based decomposition techniques [33; 34]. The drawback of this approach is that state-of-the-art BDD-based decomposition techniques cannot ensure a simpler approximate logic circuit even though the BDD for the approximate function is smaller than the BDD for the original

specification. BDD-based decomposition techniques have only recently gained attention and are an active area of research today. A common drawback of both these bottom-up approaches is that they do not solve the problem of exploring inter-dependent don't care spaces that arises in bi-directional approximations.

- Top-down approach is not flexible: In contrast to the bottom-up approach, which starts the synthesis of an approximate logic circuit with a functional specification, a top-down approach starts with a circuit implementation of the given specification. Logic optimization algorithms such as logic rewiring, recursive learning [24] that use a top-down approach make functionality-preserving modifications to the circuit structure. A similar top-down approach for synthesis of approximate logic circuits requires tracking the effects of local changes in the circuit structure and logic function on the logic functionality at the outputs. Tracking such changes is a computationally expensive, especially when seeking an approximate logic circuit that targets a specified input sub-space. Hence, the limitation of using a top-down approach for synthesis of approximate logic circuits is the the lack of flexibility in exploring the different approximate logic functions.

To address the issues of scalability and flexibility, we adopted an intermediate approach of partially clustering logic gates in the given netlist to obtain a clustered technology-independent network. A clustered technology-independent network is an intermediate representation of a circuit in which the internal nodes are Boolean func-

tions with 10–15 inputs. Each node in the clustered technology-independent network is associated with two kinds of Boolean functions: (i) a local Boolean function with variables as the inputs to the clustered node and (ii) a global Boolean function with variables as the primary inputs to the circuit. We represent the local Boolean function of a node as a sum-of-product (SOP) expression. The SOP for the off-set is referred to as the 0-SOP and the SOP for the on-set is referred to as the 1-SOP. A 0-SOP can be converted to a 1-SOP and vice-versa using DeMorgan’s law.

Some state-of-the-art logic synthesis tools today also use clustering primarily to reduce computational complexity for logic optimization. Clustering offers opportunities for technology-independent logic optimization based on local transformations, e.g., algebraic factorization, re-substitution, re-factoring [35] or global transformation, e.g., permissible functions [36]. However, the important distinction is that logic synthesis tools transformation the Boolean functions of the clustered nodes to preserve the Boolean function at the primary outputs, but approximate logic circuit synthesis algorithms must be able to simplify the Boolean function of the clustered nodes to generate one of exponentially many approximations that has a small delay and power footprint and that predicts inputs from the specified input sub-space. The next section describes the various attempts that evolved into simplification constraints that form the backbone of the synthesis algorithms for approximate logic circuits.



### 3.2 Motivation for proposed synthesis algorithm

Our initial work was aimed at developing a low overhead logic design technique for improving the reliability of a circuit. This chapter begins by describing initial attempts at applying a well-known logic design approach, known to improve area and testability of a circuit, to improve the reliability of a circuit. The limitations of this technique have served as a motivation that led us to a synthesis algorithm for unidirectional approximations, described in Sec. 3.4. This algorithm was later extended for synthesizing bidirectional approximate logic circuits as described in Sec. 3.5.

Motivated by various circuit-based implication techniques that have been used in literature for improving area and testability of a circuit, we began exploring circuit-based implications as a potential solution for improving the reliability of a circuit. Given a logic circuit, an implication relation between Boolean functions of two gates within the circuit can be leveraged to improve the reliability. If  $f_1$  and  $f_2$  are the Boolean functions of two gates,  $c_1$  and  $c_2$ , such that  $f_1 \Rightarrow f_2$ , then  $f_1$  can be used to predict the correct value of  $f_2$ , when  $f_1 = 1$ , thus protecting output of gate  $c_1$  against  $1 \rightarrow 0$  errors. The circuit structure can be modified to improve the reliability at the output of gate  $c_2$  by, replacing  $c_2$  with  $c_1 + c_2$ .

For most benchmark circuits, we found that the improvements in reliability obtained using this approach were small. Further analysis revealed that the reason for small reliability improvements was due to the inverse relation of the quality of the implication and the topological separation of the gate for the implicant function and

implied function in the given circuit. When the topological separation was large, the quality of implication was low, i.e., the implicant function covered only a small subset of the implied function, and thus, reliability improvements were small. On the other hand, when topological separation was small, the gates corresponding to the implicant and implication functions were susceptible to a large number of common failures, hence little reliability improvements were obtained by using the implicant function to predict the implied function, even with a high quality implication. Thus, we concluded that to obtain better improvements in reliability, search for implication relations will have to be extended beyond just the Boolean functions represented by the gates in the given circuit.

This motivated us to explore techniques for generating implications using extra logic gates that are not part of the given circuit. In general, this problem can be defined as searching for  $m$  gates or primary inputs in a circuit with Boolean functions  $f_1, f_2, \dots, f_m$ , such that there is a Boolean function  $h : B^{m-1} \rightarrow B$  that satisfies  $h(f_1, f_2, \dots, f_{m-1}) \Rightarrow f_m$ . Our first approach to this problem was based on clustering the gates in the circuit into nodes with 10–15 inputs, followed by generating implications using extra logic gates to protect the output of each node. To maximize reliability improvement, the outputs of the least reliable gates in the circuit were favored by the clustering algorithm to be the outputs of the clustered nodes. An implication for a node was obtained by selecting a subset of the product terms from the sum-of-products (SOP) expression of the Boolean function for each node. The re-

liability improvement obtained using this approach was better than the circuit-based implications approach. The technique was also scalable to large circuits since clustering was restricted to Boolean functions with 10–15 inputs. However, the delay overhead incurred by this technique was high due to the following reasons:

- Addition of an OR or AND gate at the output of each clustered node for improving the reliability.
- Increase in fanout of gates that serve as inputs to the extra logic gates used to generate the implications.

To reduce the delay overhead, we attempted generating implications for the primary outputs of the circuit directly using primary inputs of the circuit, thus eliminating the need for an OR or AND gate at the output of each cluster. However, the SOP-based technique used to generate implications for each cluster is not scalable to generate implications for large circuits. Traditional logic synthesis techniques are also not applicable for reasons described in Sec. 3.1. This motivated us to adopt a technique based on simplifying the Boolean functions of the nodes in a cluster with certain constraints. By imposing appropriate constraints, different clusters can be merged to obtain a completely independent logic circuit that is an implication for the Boolean function of the primary output. The remainder of this chapter describes simplification constraints used in the synthesis algorithms for unidirectional and bidirectional approximations.

### 3.3 Proposed synthesis algorithm

Given logic circuit  $C$  with  $n$  inputs and  $m$  outputs and a specified input space  $S_i$  for output  $i$ , a clustered technology-independent network,  $T$ , of  $C$  can be obtained by clustering gates in  $C$  [32]. We have used the clustering algorithms implemented in the “renode” command in ABC [37] — the open source synthesis tool. The clustering algorithm in ABC uses either SOPs or BDDs to store the Boolean function of the internal nodes. The clustering decisions are made based on two input parameters: (i) maximum number of inputs and (ii) maximum size of the Boolean function of the clustered node. The size of the Boolean function is measured as the number of cubes in the SOP if the Boolean function is stored as an SOP and as the number of nodes in the BDD if the Boolean function is stored as a BDD. We have observed that using a clustering algorithm based on storing the Boolean functions as an SOP with 10–15 inputs and a maximum size of 100 cubes in the SOP yields approximate logic circuits with small area-power-delay footprint. A possible explanation for this behavior could be that SOPs provide a better correlation to the delay and area of a circuit than BDDs.

The given logic circuit,  $C$ , is then reduced to an approximate logic circuit with a small delay-power-area footprint by simplifying the Boolean functions of the nodes in the clustered technology-independent network  $T$ . The simplification of the Boolean function is performed by selecting a subset of cubes from its 0-SOP or 1-SOP or both based on the cube weights. The cube weight indicates the importance of each cube in

predicting the output  $i$  correctly for inputs in the specified input space  $S_i$ . A weight is assigned to each cube in the 0-SOP and 1-SOP of each node  $n_j$  in the clustered technology-independent network using  $S_i$  of outputs  $i$  that contain node  $n_j$  in their fanin cone.

**Cube weight computation:** The cube weight is computed as the projection of these  $S_i$ s into the local Boolean input space of a clustered technology-independent node defines the weight of a cube. In other words, the weight of a cubes represents the fraction of minterms in  $S_i$  that will be predicted incorrectly if this cube is discarded during simplification of the clustered technology-independent node. Note that two cubes in a SOP may represent the same minterms from  $S_i$ . Thus, to avoid the same minterm from being included in the weight of more than one cube, the cubes are arranged in the increasing order of the size of their support sets and the weight of each cube is computed as the fraction of minterms in  $S_i$  that are not included in the previous cubes. The increasing support size ordering of cube weights is done to ensure that cubes with smaller support sizes are given a higher preference during cube selection to reduce the area-power-delay footprint of the approximate circuit. The cube weights are used to guide the approximation to predict the output  $i$  of the circuit correctly for inputs in  $S_i$ . To ensure correctness, additional constraints have to be imposed during simplification of clustered nodes for unidirectional approximations and extra logic has to be added to the clustered technology-independent network for bidirectional approximations.

### 3.4 Unidirectional approximation

The algorithm for synthesis of unidirectional approximate logic functions is divided into 2 stages: (i) type assignment: Assigning a type of approximation (0/1/EX/DC) to each node in  $T$ , and (ii) cube selection: Reducing the nodes in  $T$  by selecting cubes from 0-SOP, 1-SOP, or both.

#### 3.4.1 Type assignment

The aim of type assignment is to determine the type of approximation at each node in  $T$  based on the type of approximation that is desired at the primary outputs of the circuit. Local observability values are used for type assignment. For each node  $n_j$  in the multi-level network, the local observability of the fanin nodes of  $n_j$  are computed with respect to the output of  $n_j$ . The *local 0(1)-observability of a fanin node* is defined as the probability that a 0 (1) value at the fanin is observable at the output of  $n_j$ . The reason behind assigning a type based on local observability values is that if the a fanin 0(1)-observability of a fanin is dominant, then a 0(1)-approximation of the fanin would ensure a better approximation of node  $n_j$ .

Each node in  $T$  can be assigned one of 4 types: 0, 1, EX, or DC. First, for a primary output  $i$ , the node driving output  $i$  is assigned the same type as primary output  $i$ . Then, the other nodes of  $T$  are assigned a type in the reverse topological order, i.e., a node is assigned a type after all its fanout nodes have been assigned a type. The fanin of each node is assigned a type based on the local 0-observability and

1-observability of the fanin nodes. If both local 0-observability and 1-observability of a fanin node are small as compared to the observabilities of other fanin nodes, a type DC is assigned to the fanin node. If the local 0(1)-observability is greater than the local 1(0)-observability, then a type 0(1) is assigned to fanin node. If the local 0-observability and 1-observability are equal, then a type EX is assigned to the fanin node. Note that a node with more than one fanout may be assigned a different type by each fanout node. In that case, the type is assigned to the node based on the preference order of type  $EX > \text{type } 0/1 > \text{type } DC$ . Further, if a node is assigned a type 0 by one fanout and a type 1 by another fanout, then the node is assigned a type EX.

### 3.4.2 Cube selection

The goal of cube selection is two fold: (i) to ensure correctness of approximation at the primary outputs and (ii) to achieve a high approximation percentage for low overhead. Two linear time complexity algorithms for cube selection — exact cube selection and observability don't care based cube selection — are described. Exact cube selection approach guarantees correctness of the approximation at the primary outputs, but may limit the approximation percentage because strict constraints for selecting cubes are imposed to guarantee correctness. Observability don't care based cube selection relaxes the constraints for cube selection using local observability don't cares. However, this may result in an incorrect approximation at the primary out-

puts. Finally, we describe an iterative cube selection algorithm that uses exact cube selection and observability don't care based cube selection to iteratively converge to a correct approximation.

**Exact cube selection:** This technique derives an approximate logic function by picking a subset of cubes from the SOP expression of type 0 and type 1 nodes while type EX and type DC nodes are not reduced. First, the SOP expression used for cube selection must match the node type, i.e., if the node type is 0, then the cubes from the 0-SOP are selected. Cubes that conform to the fanin node types may be selected from the SOP expression. A cube is said to conform to a fanin node of type 0(1) if the literal in the cube corresponding to the fanin node is a '0'('1') or '-' (don't care). A cube conforms to a fanin node of type DC if the corresponding literal in the cube is '-'. Every cube conforms to a fanin node of type EX. A cube is selected only if it conforms to the type assignment of every fanin node. The following theorem proves that selection of cubes based on this criteria always generates a correct approximation at the primary outputs.

**Theorem:** Given Boolean functions  $X_1, X_2, g = X_1X_2$ , and on-set unidirectional approximate Boolean functions  $X'_1, X'_2$  for  $X_1$  and  $X_2$ , then  $g' = X'_1X'_2$  is a on-set unidirectional approximation for  $g$ .

**Proof:** Since  $X'_1, X'_2$  are on-set unidirectional approximations for  $X_1, X_2$ ,  $X'_1 \Rightarrow X_1$



and  $X'_2 \Rightarrow X_2$ . Thus,  $\overline{X'_1} + X_1 = 1$  and  $\overline{X'_2} + X_2 = 1$

$$\begin{aligned}
&\Leftrightarrow (\overline{X'_1} + X_1)(\overline{X'_2} + X_2) = 1 \\
&\Leftrightarrow \overline{X'_1} \overline{X'_2} + \overline{X'_1} X_2 + \overline{X'_2} X_1 + X_1 X_2 = 1 \\
&\Leftrightarrow \overline{X'_1}(\overline{X'_2} + X_2) + \overline{X'_2}(\overline{X'_1} + X_1) + X_1 X_2 = 1 \\
&\Leftrightarrow \overline{X'_1} + \overline{X'_2} + X_1 X_2 = 1 \\
&\Leftrightarrow \overline{X'_1 X'_2} + X_1 X_2 = 1 \\
&\Leftrightarrow X'_1 X'_2 \Rightarrow X_1 X_2 \\
&\Leftrightarrow g' \Rightarrow g
\end{aligned}$$

In other words,  $g'$  is an on-set unidirectional approximation of  $g$ . Similarly, we can prove that if  $g = X_1 + X_2$  then  $g' = X'_1 + X'_2$  is an on-set unidirectional approximation of  $g$ . Note that both the above results also hold true for off-set unidirectional approximations, i.e., if  $X'_1, X'_2$  are off-set unidirectional approximations of  $X_1, X_2$  then (i)  $g' = \overline{X'_1} \overline{X'_2}$  is an off-set unidirectional approximation of  $g = X_1 X_2$ , and (ii)  $g' = \overline{X'_1} + \overline{X'_2}$  is an off-set unidirectional approximation of  $g = X_1 + X_2$ . The above theorems can be generalized to  $n$  variables using induction on  $n$ .

**Observability don't-care-based selection:** The constraint for selecting cubes in exact cube selection was based on conformity of the cubes to the fanin node types. Although this constraint guarantees correctness, it limits the quality of the approximation that can be achieved. The constraint for cube selection can be relaxed by

using local observability don't cares to expand the space from which cubes can be selected. Local observability don't cares refers to the observability don't care space with respect to the output of the node, and not with respect to the primary output of the circuit. Equation 3.1 shows the computation of the Boolean space based on local observability don't cares from which cubes are selected. For simplicity, the Boolean space for a node  $n_j$  with two fanin nodes  $f_1$  of type 1 and  $f_2$  type 0 is shown.

$$\begin{aligned} g \cdot (f_1 + \overline{o(f_1)})(\overline{f_2} + \overline{o(f_2)}) & \quad \text{if } g \text{ is of type 1} \\ \overline{g} \cdot (f_1 + \overline{o(f_1)})(\overline{f_2} + \overline{o(f_2)}) & \quad \text{if } g \text{ is of type 0} \end{aligned} \tag{3.1}$$

Here,  $g$  is the local Boolean function of the node  $n_j$  and  $o(f_1)$  and  $o(f_2)$  represent the local observability of fanin nodes  $f_1$  and  $f_2$ . The Boolean space  $(f_1 + \overline{o(f_1)})$  represents the space that either conforms to the type 1 fanin node  $f_1$  or a space in which  $f_1$  is not observable. For a node of type DC, only the observability don't care term is used. Approximating the nodes in  $T$  causes incorrect values of nodes for portions of the input space. As long as only a single input of a node is incorrect, this observability don't care based cube selection ensures correctness. However, when multiple inputs of a node are incorrect, the approximation of the node may be incorrect because the observability don't care space for multiple inputs is computed as the Boolean AND of the observability don't care space for each input in Eqn. 3.1.

**Iterative cube selection algorithm:** The exact cube selection algorithm guarantees correctness of the approximation by imposing strict constraints for cube selection,

thus affecting the quality of the approximation. On the other hand, the observability don't care based cube selection relaxes the constraints on cube selection but does not guarantee correctness of the approximation. We now describe an iterative approach that combines these two techniques to achieve good quality approximations while maintaining correctness.

The SOP of every node is reduced by discarding cubes with least weights. The 0(1)-SOP is used for type 0)1) nodes. For type EX and type DC nodes, either the 0-SOP or the 1-SOP can be used. The primary outputs of the circuit are then checked for correctness of approximate functions. This can be done very efficiently using SAT algorithms, or by checking the implication condition for using BDDs. If all the outputs have been correctly approximated, the algorithm terminates. Otherwise, the outputs that have been incorrectly approximated are corrected as follows. First, a backward traversal of the circuit is performed to identify a source node of incorrect approximation. A node is a source of incorrect approximation if the Boolean function of the node has been incorrectly approximated but all its fanin nodes have been correctly approximated. The approximation of this node is corrected by using observability don't care based cube selection. If this fails to correct the approximation at the node, the exact cube selection approach is used, which guarantees a correct approximation. This procedure is repeated until the approximation at the output is fixed. The iterative cube selection algorithm flow is shown in Fig. 3.1. Note that using this iterative cube selection algorithm, it is possible that some internal nodes that

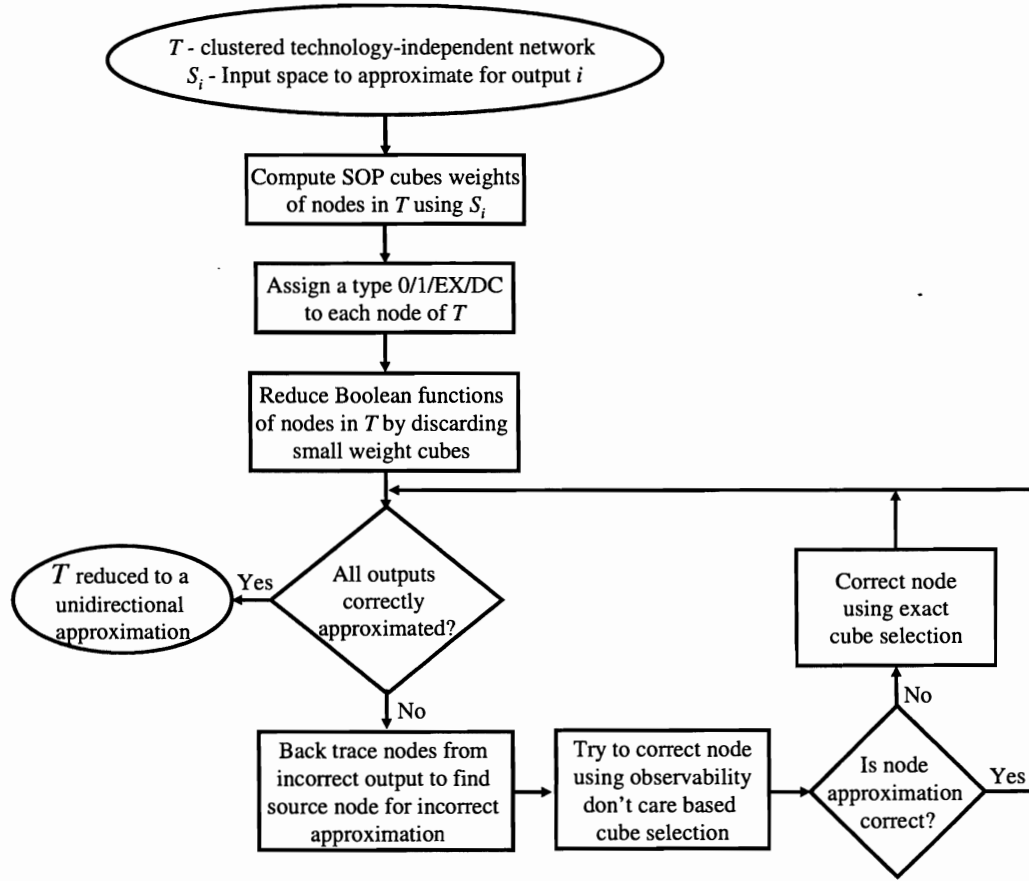


Figure 3.1 : Algorithm for synthesis of a unidirectional approximation.

are not approximated correctly, but the primary outputs are approximated correctly. Thus, the algorithm is implicitly able to explore the global observability don't care space of the internal nodes. The computational complexity of the iterative algorithm depends on the amount of backtracking that needs to be performed in order to ensure correctness of approximation. For most benchmark circuits considered, we found that no backtracking was necessary to fix the approximation at the outputs.

### 3.5 Bidirectional approximation

A simple extension of the synthesis algorithm for unidirectional approximate logic circuits is to synthesize a bidirectional approximate logic circuit using two unidirectional approximate logic circuits — one for the off-set and the other for the on-set. Logic sharing between the two unidirectional approximate logic circuits can be used to reduce the overhead of the bidirectional approximate logic circuit. However, we found that this approach for synthesizing bidirectional approximate logic circuits results in a high overhead since it fails to explore the rich interdependent don't care space in bidirectional approximations as described in Sec. 2.3. This section describes the synthesis algorithm for a predictor-indicator bidirectional approximate logic circuit by simplifying the clustered technology-independent network  $T$ . In addition to simplifying the Boolean functions of the internal nodes in the clustered technology-independent network, the algorithm also adds extra logic gates to ensure a correct predictor-indicator bidirectional approximation.

Denote  $n_j$  as an internal node in the technology independent network. First, the cube weights for the 0-SOP and 1-SOP of  $n_j$  are computed using the sum of the SPCFs of all outputs in the fanout cone of  $n_j$ . The Boolean expression of  $n_j$  is then simplified by eliminating zero weight cubes from the 0-SOP and 1-SOP of  $n_j$  to obtain the reduced on-set ( $n_j^0$ ) and the reduced off-set ( $n_j^1$ ). Using  $n_j^0$  and  $n_j^1$ , the predictor output  $\tilde{n}_j$  and indicator output  $e_{n_j}$  for node  $n_j$  are obtained as follows.

1. The reduced Boolean expressions  $n_j^0$  or  $n_j^1$  can be used as the predictor function

for  $n_j$ . The indicator function can be obtained by combining  $n_j^0$  and  $n_j^1$  using an OR gate. Thus,

$$\begin{aligned}\tilde{n}_j &= \overline{n_j^0} \text{ or } \tilde{n}_j = n_j^1 \\ e_{n_j} &= n_j^1 + \overline{n_j^0}\end{aligned}\tag{3.2}$$

2. The predictor and indicator functions for node  $n_j$  can also be obtained by exploring the interdependent don't care space as follows. First, the predictor function is set to either  $\overline{n_j^0}$  or  $n_j^1$  and the indicator function is set to  $n_j^1 + \overline{n_j^0}$ . Next, the predictor function,  $\tilde{n}_j$ , can be optimized using the Boolean space  $e_j = 0$  as the don't care space. Further, the indicator function  $e_j$  can be optimized by using the Boolean space  $\tilde{n}_j = n_j$  as the don't care space. This procedure is repeated to generate unique pairs of predictor and indicator functions.

Among the various predictor-indicator functions, the pair with the least number of literals in the SOP expressions of the predictor and indicator is chosen. The output  $e_{n_j}$  is 1 when an inputs from the SPCF of any output in the fanout cone of  $n_j$  is applied and output  $\tilde{n}_j$  predicts the correct value of  $n_j$  when  $e_{n_j}$  is 1. The indicator output  $e_i$  for primary output  $i$  is 1 when all internal nodes in the fanin cone of primary output  $i$  predict their outputs correctly. Thus,  $e_i$  can be generated as a Boolean AND of the indicator outputs,  $e_{n_j}$ , of all internal nodes  $n_j$  in the fanin cone of output  $i$ . The simplified technology-independent network  $\tilde{T}$  is then synthesized, optimized, and

mapped obtain a bidirectional approximation for the output  $i$ . The pseudo code for synthesis of a bidirectional approximate circuit is presented in Algorithm 1.

---

**Algorithm 1:** Bidirectional( $T, S_i$ )

---

**input** : Clustered technology-independent network  $T$  and specified input space  $S_i$  of output  $i$   
**output** : Reduced technology-independent network of the bidirectional approximation

Compute cube weights for the 0-SOP and 1-SOP of all nodes in  $T$

**foreach** node  $n_j$  in  $T$  **do**

- Remove zero weight cubes from 0-SOP of  $n_j$  to obtain  $n_j^0$
- Remove zero weight cubes from 1-SOP of  $n_j$  to obtain  $n_j^1$
- $L = \{\}$  /\* list of unique  $\{n_j, e_{n_j}\}$  \*/
- Assign  $\tilde{n}_j = \overline{n_j^0}$  and  $e_{n_j} = n_j^1 + \overline{n_j^0}$
- repeat**
  - Minimize SOP of  $\tilde{n}_j$  using  $e_{n_j} =$  as the don't care space
  - if**  $\{\tilde{n}_j, e_{n_j}\}$  is unique **then**
    - └ Add  $\{\tilde{n}_j, e_{n_j}\}$  to  $L$
  - Minimize SOP of  $e_{n_j}$  using  $\tilde{n}_j = n_j$  as the don't care space
  - if**  $\{\tilde{n}_j, e_{n_j}\}$  is unique **then**
    - └ Add  $\{\tilde{n}_j, e_{n_j}\}$  to  $L$
- until** unique  $\{\tilde{n}_j, e_{n_j}\}$
- Assign  $\tilde{n}_j = n_j^1$  and  $e_{n_j} = n_j^1 + \overline{n_j^0}$
- repeat**
  - Minimize SOP of  $\tilde{n}_j$  using  $e_{n_j} =$  as the don't care space
  - if**  $\{\tilde{n}_j, e_{n_j}\}$  is unique **then**
    - └ Add  $\{\tilde{n}_j, e_{n_j}\}$  to  $L$
  - Minimize SOP of  $e_{n_j}$  using  $\tilde{n}_j = n_j$  as the don't care space
  - if**  $\{\tilde{n}_j, e_{n_j}\}$  is unique **then**
    - └ Add  $\{\tilde{n}_j, e_{n_j}\}$  to  $L$
- until** unique  $\{\tilde{n}_j, e_{n_j}\}$
- Pick  $\{\tilde{n}_j, e_{n_j}\}$  from  $L$  with the minimum total literals in the two SOPs

Output of  $\tilde{T}$ , reduced from the clustered technology-independent network  $T$ , is the predictor of the bidirectional approximation

AND of all  $e_{n_j}$ s in  $\tilde{T}$  is the indicator of the bidirectional approximation

---

The framework for synthesizing approximate logic circuits was implemented in ABC, the logic synthesis tool developed at Berkeley [37]. The remainder of this thesis describes how the algorithms described in this chapter can be used to synthesize approximate logic circuits for different applications by changing the specified input space  $S$ . Chapter 4 demonstrates error resilient design based on concurrent error detection/masking using approximate circuits. Chapter 5 proposes new flip-flop designs

to reduce combinational logic overhead for masking timing errors based on approximate circuits. Chapter 6 proposes a new logic decomposition technique based on approximate circuits for optimizing performance during logic synthesis. Chapter 7 proposes a new variable partition algorithm for bi-decomposition of large Boolean functions.



## Chapter 4

### Improving reliability with approximate circuits

As described in Chapter 1, due to the increasing variability with technology scaling, reliability has emerged as a serious concern in CMOS designs. In recent years, since post-manufacturing test and burn-in techniques becoming less effective in ensuring reliable operation during the lifetime of a chip, there has been a significant interest in exploring novel circuit and logic design solutions for improving reliability of logic circuits.

Reliable design solutions are already used in commercial chips today to detect/correct errors in parts of the system that are most vulnerable to failures. For instance, single error correction/double error detection (SEC/DED) codes are used to protect semiconductor memories against bit flips. For high-reliability or mission-critical applications, residue logic is used to protect data-path logic circuits such as the arithmetic unit against errors arising due to transient and permanent failures. As variability and failures increase with technology scaling, parts of the design, e.g., logic and flip-flops on the microprocessor control path, that were earlier considered robust are also becoming vulnerable to failures. Due to the irregular structure, reliable design of control logic circuits is challenging and hence, recent research has focused on developing cost-effective reliable design solutions for irregular multi-level logic circuits. Reliable

design solutions mainly target two kinds of errors in logic circuits.

- Logical errors: Intermittent failures arising due to latent manufacturing defects and transient failures arising due to external factors, e.g., single-event upsets due to radiations strikes, when propagated through a logic circuit, can result in an error at the output of the logic circuit. Since these errors are not timing-related and cannot be avoided by reducing the clock frequency, they are referred to as “logical errors”.
- Timing errors: Delay variations due to temperature/supply voltage variations and long-term aging effects like NBTI, PBTI, and TDDB can result in an incorrect value to be latched at the output of a circuit. Since these errors are timing-related and can be avoided by reducing frequency of operation, they are referred to as “timing errors”.

The remainder of this chapter describes the application of approximate logic circuits to detect/mask logical and timing errors.

## 4.1 Design solution for logical errors

This section describes two commonly used techniques — concurrent error detection (CED) and concurrent error masking (CEM) — in literature for improving reliability of logic circuits to logical errors. The term “concurrent” refers to the ability of these techniques to dynamically detect or mask errors arising during normal operation of a logic circuit.

#### 4.1.1 Concurrent error detection

Concurrent error detection (CED) has been used to detect faults in systems where dependability and data integrity are of importance [38; 6; 39; 40; 7; 41; 42]. Classical CED techniques focused on guaranteeing 100% coverage of broad classes of errors and generally incurred a performance penalty along with a large area and power overhead (often in excess of 100%), especially for irregular control logic circuits. Recent research has seen the emergence of low overhead CED techniques that seek to meet coverage requirements at minimum cost, e.g., [9; 43]. However, these techniques usually target a single fault model such as stuck-at faults or single-event upsets, and cannot be customized to a broad class of failure mechanism arising due to various process and dynamic variability effects. There are several disadvantages of these techniques including limited scalability, lack of options to trade-off coverage for overhead, and requiring modifications to or constraining synthesis of the original design.

Unlike CED techniques such as [7; 41; 42; 10], CED using approximate circuits is non-intrusive, i.e., it does not require any modification in the synthesis of the original logic circuit. Approximate circuits are also designed to have a smaller delay than the original logic circuit, and hence approximate circuits incur negligible performance penalty. The hardware overhead of the approximate circuit can also be traded-off for the error detection coverage. Using prior knowledge of the input vector distribution, either acquired online or using cycle accurate simulations, an approximation can also be tailored to provide better and targeted error masking coverage for the same

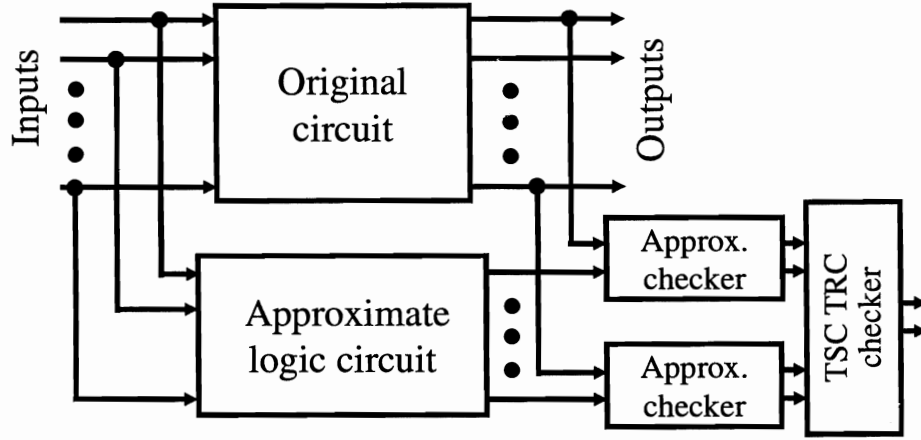


Figure 4.1 : CED based on approximate logic circuits

overhead.

This section proposes a low overhead, non-intrusive solution for CED based on approximate logic circuits. Since CED is non-intrusive, no modifications are necessary to the original design. The synthesis algorithm for approximate logic circuits, as presented in Chapter 3, also provides fine-grained trade-offs between area-power overhead and CED coverage. A self-checking checker that is compatible with the proposed CED technique is also proposed. The checker produces two-rail encoded outputs, ensuring compatibility with other error detection techniques for error signal consolidation.

CED based on approximate logic circuits is illustrated in Fig. 4.1. Just as in conventional CED, the approximate logic circuit is used as the check symbol generator for the given logic circuit. For every primary output of the circuit, a unidirectional approximate logic circuit (either for the on-set or for the off-set) is used for detection

of  $1 \rightarrow 0$  or  $0 \rightarrow 1$  errors. The type of approximation for a primary output is decided by the type of error ( $1 \rightarrow 0$  or  $0 \rightarrow 1$ ) that dominates at that output. For each primary output, the appropriate 0/1-approximate logic circuit is synthesized and its corresponding output in the approximate logic circuit is checked using the approximate checker. The outputs of the approximate checkers are consolidated using a conventional totally self-checking (TSC) two-rail code (TRC) checker [44].

**Totally self-checking checker:** Checker design is an integral part of concurrent error detection. The function of the checker is to monitor the output of the circuit and the check symbol generator, and to signal an error when they do not form a valid codeword. Checkers are usually designed to be totally self-checking (TSC) by satisfying the code-disjoint, fault-secure, and self-testing properties w.r.t a specified fault class. Consider an output  $y$  that has a unidirectional approximate circuit for the off-set (because a  $0 \rightarrow 1$  error is dominant) for error detection. Denote the output of the unidirectional approximate circuit for the off-set of  $y$  by  $z$ . By definition of a unidirectional approximation,  $\bar{z} \Rightarrow \bar{y}$  and  $y \Rightarrow z$ . Thus, when  $z=0$ , the approximate logic circuit detects  $0 \rightarrow 1$  errors at  $y$ , and CED is active. A small fraction of undetected errors arise when  $z=1$ , and CED is not active. The proposed approximate checker, shown in Fig. 4.2 (b), is TSC w.r.t all single stuck-at faults when CED is active. When CED is not active, there are exceptions where the checker violates the TSC property.

Code-disjointness ensures that the checker gives an invalid output codeword when

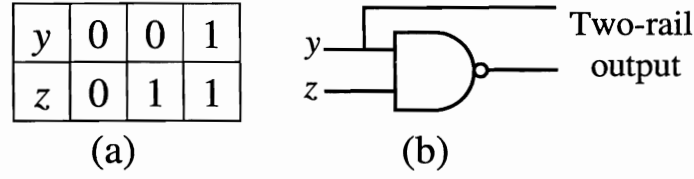


Figure 4.2 : Totally self-checking checker design

an invalid codeword is presented at its inputs. The valid input codeword space for the checker is shown in Fig. 4.2(a). Since,  $z$  is a unidirectional approximation for the off-set of  $y$ , the input codeword space does not contain  $z = 0, y = 1$ . The output codeword space is the two-rail code, i.e.,  $\{01, 10\}$ . This is desirable because the outputs of the checkers can be consolidated using a two-rail code checker. It is evident from Fig. 4.2(a) that the checker is code-disjoint.

Self-testing ensures that all the faults in the specified fault class are testable under normal operation. Since CED with approximate logic circuits protects only unidirectional errors, errors due to faults in the unprotected direction ( $y$  stuck-at 0 and  $z$  stuck-at 1 for an off-set unidirectional approximation) violates the self-testing property. Since  $z$  is an off-set unidirectional approximation of  $y$ , the fault  $y$  stuck-at 0 will always violate the self-testing property as  $z=1, y=1$  is the only input vector that can test  $y$  stuck-at 0. However, for  $y$  stuck-at-0, the input to the checker becomes  $z=1, y=0$ , which is a valid codeword and so cannot detect the fault. Hence, no input vector under normal operation can detect  $y$  stuck-at 0. Similarly, since  $y$  is an on-set unidirectional approximation of  $z$  when  $z$  is an off-set unidirectional approximation of  $y$ , the fault  $z$  stuck-at-1 will always violate the self-testing property. Note that

since the checker is an irredundant logic circuit, it is completely testable offline for all single stuck-at faults through incorporation into a scan chain.

Fault-secureness ensures that a fault within the fault class either gives the fault-free response or an invalid codeword at the output of the checker. The checker maps an asymmetric input codeword space to the dual-rail code-space at the output. Since  $z$  is an off-set unidirectional approximation of  $y$ , the checker is not fault secure for stuck-at faults at  $y$  when  $z=1$ .

Although the discussion so far has focused on using approximate logic circuits for non-intrusive CED, further reductions in area-power overhead can be achieved by merging structurally or functionally equivalent nodes between the original and the approximate logic circuit. Thus, it is possible to trade-off CED coverage for further reductions in overhead. However, sharing of logic between the original and approximate logic circuits makes CED intrusive. Partial duplication for CED described in [9] can be viewed as a special case of approximate logic functions with logic sharing. In partial duplication, the approximate logic function has a 100% approximation percentage and non-critical nodes are shared between the original and the approximate logic circuits.

The results in Table 4.1 and Table 4.4 are reported for the single fault model with all the gates in the circuit having the same probability of failure. Using this fault model, the circuit is simulated with a randomly injected single stuck-at fault and a random input vector, for 6.4M runs. CED coverage is the percentage of runs for

which an error at the output is detected by the CED technique. The inputs to the circuits are assumed to have an equal probability of occurrence, i.e., there is no input vector biasing.

Table 4.1 shows results for single output cones extracted from MCNC benchmarks. The first column is the name of the benchmark circuit from which the output cone was extracted. The second column is the number of gates in the output cone. The third column is the area overhead of the synthesized approximate logic circuit. The fourth column is the approximation percentage achieved by the approximate logic function. The fifth column is the maximum error detection percentage that can be achieved by protecting the dominant error ( $0 \rightarrow 1$  or  $1 \rightarrow 0$ ) at the output. The sixth column reports the error detection percentage, i.e., the CED coverage achieved by the synthesized approximate logic circuit. The results illustrate the effectiveness of the synthesized approximate logic functions in achieving a high approximation percentage for low area overhead. The disparity in the approximation percentage and CED coverage for circuits `des` and `i8` is because CED coverage is limited by the amount of skew in the type of errors ( $0 \rightarrow 1$  vs.  $1 \rightarrow 0$ ) at the output.

Table 4.1 : Approximation percentage and CED coverage for output cones extracted from benchmark circuits.

Name	Gates	Area (%)	Approx. (%)	CED coverage (%)	
				Max.	Achieved
i8	106	28	80	65	50
des	191	2.7	95.6	56	48
dalv	862	25	93.8	85	71
i10	1141	1.5	91	76	64



The results for error detection in complete MCNC benchmark circuits are shown in Table 4.4. The proposed synthesis algorithms for approximate logic functions were evaluated on logic benchmarks from this suite. Logic benchmarks with a reasonably large skew in the errors at the outputs have been chosen. Three metrics — area, power overhead and CED coverage have been chosen for comparing the proposed CED technique with existing CED approaches. Area is computed as the total number of gates in the circuit, power is computed as the total switching activity of the gates in the circuit, and CED coverage is computed using fault injection and simulation as described above. Area, power overhead and CED coverage for completely non-

Table 4.2 : Area-power overhead and CED coverage for MCNC benchmark circuits.

Name	Gates	Max. CED coverage	No logic sharing			Logic sharing		Partial duplication			Parity prediction		
			Area	Power	CED coverage	Area	CED coverage	Area	Power	CED coverage	Area	Power	CED coverage
cmb	57	99.7	32	26	98	29	98	48	32	98	87	43	66
cordic	116	88	28	37	82	24	82	26	22	82	29	33	71
term1	260	82	15	25	71	13	70	17	19	70	100	101	92
x1	442	78	36	45	68	26	65	30	37	68	125	120	86
i2	440	89	5	6	84	3	83	6	4	82	100	100	100
frg2	1089	90	30	47	80	22	75	46	48	79	161	133	91
dalv	1166	92	21	35	80	15	77	44	44	77	110	109	94
il0	2866	85	36	56	81	30	77	54	49	81	139	135	64

intrusive approximate logic functions are shown under the column “No logic sharing”. The trade-off between area overhead and CED coverage achieved by merging of non-critical nodes between the original and approximate logic circuit is shown under the column “Logic sharing”. These are compared to two existing approaches — intrusive CED based on partial duplication [9] and non-intrusive CED based on single-bit parity checkers. The results show that the same CED coverage can be achieved with

the proposed technique with an area overhead that is lower than that for partial duplication. Furthermore, the proposed technique is completely non-intrusive and incurs zero performance penalty because the approximate logic function always has a lower delay on the critical path. For the benchmarks studied in this thesis, the delay of the approximate logic circuit was 38% less than the delay of the original circuit on average. The proposed technique is also scalable and the runtime for the synthesis of the approximate logic circuit for the largest benchmark circuit, i10, was 5m28s. Single-bit parity prediction requires average area and power overhead of 106% and 97%, which is roughly 3X higher than the proposed solution, for a 2% improvement in CED coverage on average. In benchmark circuits cmb and i10, single-bit parity prediction has higher area and power overheads but lower CED coverage as compared to both approximate logic functions and partial duplication. In addition, single-bit parity prediction produces circuits with higher critical path delay. For the benchmark circuits used in this thesis, the critical path delay of a single-bit parity prediction circuit was 51% higher than the original circuit on average.

The error rate ( $1 \rightarrow 0$  or  $0 \rightarrow 1$ ) for each output can be computed using circuit simulation or reliability analysis algorithms [45; 46; 47]. If the error rates are technology dependent, a quick synthesis and mapping pass on the multi-level technology-independent network is used to obtain the error rates at the primary outputs. The CED coverage presented in Table 4.3 demonstrates that the CED coverage using approximate circuits is technology-independent, i.e., it is not significantly affected by

Table 4.3 : Technology-independence of CED coverage

Name	CED coverage %				
	Impln 1	Impln 2	Impln 3	Impln 4	Impln 5
cmb	95.8	96	96.6	95.1	96.7
cordic	74	74.5	74.1	74.6	73
term1	70	73	75	80	71
x1	67.8	68.6	64.1	64.5	68
i2	79	84	82	85	83
frg2	70	69	71.3	76.1	75.2
dalv	71.2	72.1	73	72.4	75
i10	70	71.2	70.5	71.7	72.2

the (i) synthesis scripts used to optimize and map the original and approximate logic circuits or (ii) the library used to map and perform reliability analysis on the original circuit.

**Technology-independence:** For each benchmark circuit, reliability analysis was performed on a netlist obtained by quick synthesis to determine the type of approximation for each output. After reliability analysis, an approximate logic circuit was synthesized to detect the dominant error ( $1 \rightarrow 0$  or  $0 \rightarrow 1$ ) at each output. Five technology-mapped implementations of the original circuit were generated using different optimization scripts in ABC and different technology libraries. The same approximate logic function (mapped with the technology library of the original circuit) was used to detect errors in each of the implementations. Table 4.3 shows the CED coverage for different technology-mapped implementations of the original and approximate logic circuits for the same area-power overheads. The table illustrates that the CED coverage remains fairly constant for different technology-mapped im-

plementations. Thus, the effectiveness of CED achieved using the proposed technique depends mainly on the Boolean function being approximated, i.e., it is technology-independent.

#### 4.1.2 Concurrent error masking

CED techniques need hardware support, e.g., roll-back or local instruction replay, to correct a detected error. Roll-back incurs significant performance penalty and local instruction replay requires extensive hardware support, especially in high-performance designs that typically have deep pipelines and complex control logic. Error masking solutions eliminate the performance and hardware overhead associated with roll-back or instruction replay by masking errors dynamically during normal operation. However, classical error masking solutions like triple modular redundancy (TMR) and nand multiplexing [5] are not useful in mainstream applications because they incur significant hardware overhead just for error masking.

Recently, partial error masking [10] of logical errors based on triplicating the most critical portions of a logic circuit have been proposed. Although this technique is power-efficient and has a low performance penalty, it is susceptible to common mode failures [23]. Resynthesis [11] and rewiring [12] techniques improve reliability by restructuring logic to increase logical masking within the combinational logic circuit. However, often the restructured logic circuit has a larger critical path delay and hence, it hurts performance. Furthermore, the improvements in reliability achieved using

local resynthesis [11] and logic rewiring [12] techniques is limited by the structure of the given logic circuit. For instance, the average improvement in reliability achieved using resynthesis [11] is 39.8% (13.1% area overhead) and using rewiring [12] is 11.8% (6.9% area overhead). The scalability of rewiring [12] is also questionable because it has been demonstrated only on circuits with 20-30 inputs and a few hundred gates.

Unlike partial error masking [10], logic rewiring [12], and resynthesis [11] techniques, concurrent error masking using approximate circuits is non-intrusive, i.e., it does not require any modification in the synthesis of the original logic circuit. Since approximate circuits for masking errors are designed to have a smaller delay than the original logic circuit and since errors are masked directly at the outputs of the original logic circuit, approximate circuits incur negligible performance penalty and are useful for designing reliable, high-performance systems. Since the approximate logic circuit is structurally significantly different from the original logic circuit, concurrent error masking based on approximate circuits eliminates common mode failures [23]. Furthermore, techniques such as [11; 12], that modify the logic structure of the original logic circuit to improve reliability of logic circuits, can potentially hurt the testability by introducing hard-to-detect faults. In contrast, non-intrusiveness allows approximate circuit to be gated during post-manufacturing test, thus preserving the testability of the given logic circuit.

Approximate logic circuits can be used to mask logical errors at an output  $y$  in the given logic circuit as follows. Using an off-set unidirectional approximation,  $\tilde{y}_i$ , of

an output,  $y_i$ ,  $0 \rightarrow 1$  errors can be masked by combining  $y_i$  and  $\tilde{y}_i$  using an and gate. Since  $\overline{\tilde{y}_i} \Rightarrow \overline{y_i}$ , when  $\tilde{y}_i$  is 0,  $y_i$  is also 0, and an error at  $y_i$  or  $\tilde{y}_i$  can be masked by the and gate. Thus, errors can be masked even when the outputs of the unidirectional approximate circuit are vulnerable to errors arising due to latent defects or single-event upsets. Similarly,  $1 \rightarrow 0$  errors can be masked by combining an output with its on-set unidirectional approximation using an or gate.

A unidirectional approximation can either mask  $1 \rightarrow 0$  (on-set unidirectional approximation) errors or  $0 \rightarrow 1$  (off-set unidirectional approximation) errors at an output of the circuit. To maximize error masking coverage, we use the unidirectional approximation that masks the higher of the two error rates. The  $1 \rightarrow 0$  and  $0 \rightarrow 1$  error rates can be computed either using Monte Carlo simulations or using reliability analysis tools, e.g., [45; 47]. Further, the specified input space  $S_i$ , to predict output  $y_i$  correctly, is the on-set of  $y_i$  for an on-set unidirectional approximation and the off-set of  $y_i$  for an off-set unidirectional approximation. The goal of the unidirectional approximate circuit is to maximize correct prediction of inputs in  $S_i$  with small overhead. This is achieved by using the synthesis algorithm described in Section 3.4.

Concurrent error masking for logical errors based on unidirectional approximate logic circuits is shown in Fig. 4.3. For each output  $y_i$  of the circuit, either a 1-approximate or a 0-approximate logic circuit is used for masking of  $1 \rightarrow 0$  or  $0 \rightarrow 1$  errors. The approximate logic circuit is synthesized using the algorithm described in Sec. 6.3. Finally, error masking is performed by combining each output of the

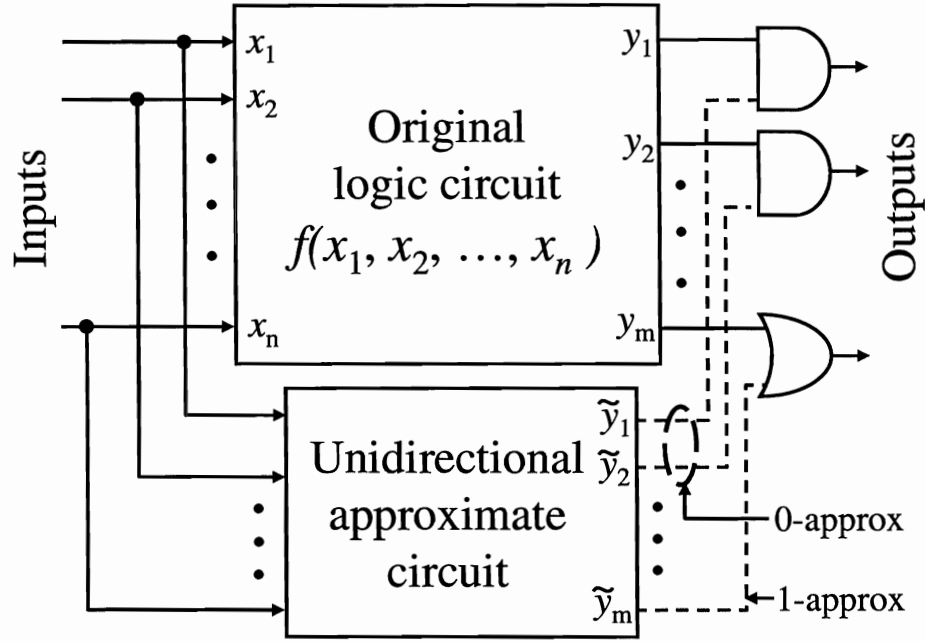


Figure 4.3 : Concurrent error masking based on approximate logic circuits.

given logic circuit with its off-set unidirectional approximation (on-set unidirectional approximation) with an **and** (**or**) gate. Error masking based on approximate circuit does not incur a performance penalty since the approximate circuit has a smaller delay. Further, approximate logic circuits allow flexible trade-offs between error masking coverage and the power/area overhead incurred by the approximate logic circuit.

Note that a bidirectional approximation,  $\hat{y}_i = \{\tilde{y}_i, e_i\}$ , of an output  $y_i$  cannot be used to mask logical errors. This is because a 2-to-1 multiplexer is used to implement error masking for a bidirectional approximation, with  $y_i$ ,  $\tilde{y}_i$ , and  $e_i$  as the 0-data, 1-data, and select inputs to the 2-to-1 multiplexer, respectively. When the indicator output,  $e_i$ , is 0,  $\tilde{y}_i$  may not predict  $y_i$  correctly, and hence errors at  $y_i$  are not masked. When  $e_i$  is 1,  $\tilde{y}_i$  predicts  $y_i$  correctly, and thus, an error at  $\tilde{y}_i$  is masked by using  $\tilde{y}_i$

instead of  $y_i$ . However, if an error occurs at  $\tilde{y}_i$  when  $e_i$  is 1, the 2-to-1 multiplexer output would be incorrect. Hence, a bidirectional approximation cannot be used when the bidirectional approximate circuit is vulnerable to errors.

Table 4.4 : Area and power overhead for concurrent masking of logical errors.

Circuit	I/O	Gates	Area	Base error rate	Max. coverage	Masking coverage	Overhead (%)		Slack (%)
							Area	Power	
cmb	16/4	20	31	$5.7 \times 10^{-2}$	99.9	94	32	11	55
x2	10/7	30	44	$7.3 \times 10^{-2}$	65	45	27	9.8	60
il	25/12	41	52	$4.1 \times 10^{-2}$	72	55	23	7.3	68
cu	14/11	44	55	$4.5 \times 10^{-2}$	86	72	40	30	10
cc	21/20	49	71	$3.5 \times 10^{-2}$	76	68	55	41	41
too_large	45/45	251	368	$9.1 \times 10^{-2}$	79.6	72	35	16	48
frg2	143/135	531	784	$3.6 \times 10^{-3}$	76.4	70	22	7.2	48
il0	257/224	1536	2189	$3.9 \times 10^{-3}$	82	78	36	20	18
sparc_ifu_dec	131/107	685	994	$2.9 \times 10^{-3}$	84.2	77	30	12	53
sparc_ifu_dcl	136/194	392	563	$7.0 \times 10^{-3}$	66.6	50.2	30	12	35
lsu_excptl <sup>†</sup>	251/44	360	510	$3.6 \times 10^{-3}$	75.9	72	36	20.3	34
sparc_ifu_errctl <sup>†</sup>	347/132	865	1260	$1.8 \times 10^{-3}$	77.1	71.3	37.2	26.4	65
<b>Average</b>	–	–	–	–	78.4	68.7	33.6	17	44.6

<sup>†</sup> These circuits contained output cones with less than 20 gates that were eliminated.

The error rate for logical errors arising due to transient failures is computed using circuit simulation. In each error rate simulation run, a randomly generated input pattern is applied to the circuit and a fault is injected at a single gate in the circuit. The inputs patterns applied to the circuits are assumed to have an equal probability of occurrence, i.e., there is no input vector biasing. The error rate for each output is the fraction of simulation runs that result in an error at that output. A circuit is simulated for 6.4M runs to compute the error rate at each primary output. For the given logic circuit, the error rate for the outputs computed using the above error rate simulation technique is designated as the base error rate. When the approximate



circuit is used as an error masking circuit, the error rate for the outputs are reduced. The reduction of the error rate relative to the base error rate is reported as the error masking coverage. Note that in a design that uses an approximate circuit for error masking, the error rate at the outputs is computed by randomly injecting a fault at one gate either in the original logic circuit or the approximate circuit.

Table 4.4 presents results for concurrent error masking of logical errors. The first four columns are the name, inputs/outputs, gate count, and area of the benchmark circuit. The fifth column in Table 4.4 reports the average base error rate, i.e., the average error rate over all primary outputs for the benchmark circuit without error masking support. Then, a unidirectional approximate circuit is synthesized to mask the type of error,  $1 \rightarrow 0$  or  $0 \rightarrow 1$ , that dominates at each primary output. Since either  $1 \rightarrow 0$  or  $0 \rightarrow 1$  errors are masked, the sixth column indicates the maximum error masking coverage that can be achieved. The last four columns indicate the actual error masking coverage, area, power overhead incurred, and timing slack of the approximate logic circuit.

Like existing partial error masking techniques [11; 12], fine-tuning the error masking coverage obtained using approximate logic circuits would require iterative error simulation. However, the error masking coverage can be controlled to a great extent in the cube selection phase during synthesis of the approximate circuit. In our implementation, the cubes for each clustered node are selected until the total weight of the selected cubes reaches a threshold of the total weight of all cubes. We use the thresh-

old parameter to converge to an area overhead of around 30% for the approximate circuit.

The results illustrate the effectiveness of the approximate logic circuits in achieving a high error masking coverage with a low delay-area-power footprint. On an average, over 12 benchmark circuits, the maximum error masking coverage is 78%. Thus, an error masking coverage of 78% can be obtained by duplicating the circuit, i.e., with an area overhead of 100%. Conventional error masking techniques such as triple modular redundancy can provide 100% error masking coverage for an area overhead of 200%. Using unidirectional approximate circuits, 88% of errors in one direction can be masked with an area (power) overhead of 34% (17%). This translates to a net error masking coverage of 69%.

## 4.2 Design solution for timing errors

Unlike design solutions for logical errors, most existing design solutions focus on detection of timing errors during normal operation.

### 4.2.1 Related work: Timing error detection and prediction

Existing techniques for timing error detection are based on resampling the outputs [18] or monitoring delayed transitions at the outputs [15; 16; 13; 48; 17] of a circuit. These techniques require roll-back or local instruction replay to correct timing errors which, as mentioned earlier, incurs a performance penalty in high-performance designs. Fur-

ther, resampling based timing error detection techniques also suffer from data path metastability and increased clock energy due to the addition of an extra latch [49].

More recently, error prediction techniques, based sensors that monitor for transitions arriving too close to the clock edge [17], canary flip-flops [20], and duplicate critical paths [50], have been proposed. However, these techniques can only predict timing errors arising due to gradual slowdown of critical paths, e.g., due to NBTI, PBTI, and TDDB, but not due to fast-changing dynamic variability effects, e.g., temperature/supply voltage variations and clock jitter. Furthermore, error prediction techniques also incur a performance penalty due to the timing guardband required for detecting slowdown of critical paths. A detailed description and comparison of existing solutions for timing error detection and prediction is presented in Chapter 5.

#### 4.2.2 Timing error masking using approximate circuits

Timing errors in a given logic circuit can be masked by synthesizing an approximate logic circuit that predicts an input sub-space for which the given logic circuit is vulnerable to timing errors. This input sub-space, referred to as the speed-path characteristic function (SPCF) in this section), is the specified input sub-space,  $S$ , for the approximation. This section first proposes an efficient algorithm for computing the SPCF and then, describes how an approximate circuit synthesized with the SPCF as the specified input sub-space,  $S$ , can be used to mask timing errors in a given logic circuit.

Table 4.5 : Accuracy vs. runtime for computing speed-path characteristic function with different approaches

Circuit	I/O	Area	Node-based [51]		Path-based extn. of [51]		Proposed short-path-based approach	
			Over-approximation		Exact		Exact	
			Crit. inputs	Runtime	Crit. inputs	Runtime	Crit. inputs	Runtime
C432	36/7	147	$4.4 \times 10^{10}$	0.82s	$3.3 \times 10^7$	4.96s	$3.3 \times 10^7$	1s
C2670	233/140	568	$9.9 \times 10^{67}$	1.1s	$8 \times 10^{66}$	1.6s	$8 \times 10^{66}$	0.58s
sparc_ifu_dec	131/146	887	$6.4 \times 10^{38}$	0.01s	$4.2 \times 10^{31}$	0.07s	$4.2 \times 10^{31}$	0s
sparc_ifu_invctl	173/115	442	$3.04 \times 10^{63}$	0.4s	$3.46 \times 10^{62}$	0.59s	$3.46 \times 10^{62}$	0.32s
lsu_stb_ctl	182/169	810	$6.7 \times 10^{52}$	0.18s	$3.8 \times 10^{50}$	0.36s	$3.8 \times 10^{50}$	0.13s

**Speed-path characteristic function (SPCF):** The SPCF for a given logic circuit is computed using technology-dependent gate and interconnect delays. This thesis describes an algorithm for computing the SPCF for a technology-mapped circuit, but for better accuracy the same algorithm can be also be used to compute the SPCF after the place-and-route of a design. Consider a technology-mapped circuit  $C$  with primary inputs  $x_1, x_2, \dots, x_n$  and primary outputs  $y_1, y_2, \dots, y_m$ . For a given primary input pattern  $I$ , the output of the circuit stabilizes to the correct value at output  $y$  after a finite, non-zero delay  $\Delta_I$ . The value of  $\Delta_I$  depends on the applied input pattern  $I$ , gate delays and the circuit structure. Given a target arrival time at output  $y$ ,  $\Delta_y$ , an input pattern  $I$  is referred to as a *speed-path activation pattern* iff  $\Delta_I > \Delta_y$ .

**Definition:** For a given target arrival  $\Delta_y$  at output  $y$ , the *speed-path characteristic function (SPCF)*, denoted by  $SPCF_y(x_1, x_2, \dots, x_n, \Delta_y)$ , is the characteristic function for the set of all speed-path activation patterns. Thus, if speed-paths within 10% of the critical path delay,  $\Delta$ , are targeted, then  $\Delta_y = 0.9\Delta$ . In the rest of this thesis,

the SPCF at  $y$  is denoted by  $SPCF_y(\Delta_y)$  for brevity.

The problem of computing the SPCF was first introduced in the context of timing-driven decomposition of logic circuits [26]. In [52], an exact algorithm for computation of the SPCF was proposed using an ADD-based timing analysis framework. However, the ADD-based approach is highly memory and time intensive, especially when a complex and realistic gate delay model is used [53]. To address the problem of computational complexity, algorithms that compute a super-set of the SPCF, instead of the exact SPCF, have been proposed [53; 51]. Results presented in [51] indicate that the approach presented in [53] may lead to large over-approximations of the SPCF for most circuits. The approach presented in [51] extends the node-based approach presented in [53] to reduce the over-approximation in the SPCF. Using arrival and required time information, gates with a negative slack are marked as critical. With the help of two functions, long path activation function and short path activation function, both static and dynamic sensitizable patterns are computed in a single topological pass through the critical gates in the circuit. The algorithm is node-based because the critical gates are marked statically, i.e., before the topological pass through the circuit. Thus, if a gate has more than one fanout and the gate lies on a critical path only along one fanout, the gate is marked critical and input patterns that sensitize any path through this gate are included in the SPCF. Thus, the over-approximation in the SPCF arises as a consequence of the node-based approach, i.e., *statically* marking critical gates *before* the topological pass to compute the SPCF.

The node-based strategy is also a major reason for the computational efficiency of the algorithm. The node-based algorithm from [51] can be extended to a path-based algorithm to compute the SPCF exactly. In a path-based approach, gates are not marked as critical based on required and arrival time information. Instead, the gates are denoted as critical in the context of the path on which it lies. However, the accuracy of the path-based extension comes at the cost of computational complexity. The trade-off between accuracy and runtime for the node-based approach of [51] and the proposed path-based approach is illustrated in Table 4.5. The first 3 columns indicate the name, input/output count and area of the circuit. The speed-path characteristic function is computed as the set of all input patterns that sensitize speed-paths within 10% of the critical path delay. The number of critical patterns, i.e., the number of input patterns in the speed-path characteristic function and the runtime for computing the set of critical patterns for the node-based approach [51] and the path-based extension are shown in columns 4 and 5 respectively. Note that the set of critical patterns computed using the node-based approach is always a super-set of the set of critical patterns computed using the path-based proposed. However, the path-based approach is, on average, 3.5X slower than the node-based approach.

The computational complexity of the path-based extension of [51] can be attributed to the path traversals for the computation of long path activation function and short path activation function. In this thesis, we show that the computational complexity can be reduced significantly by computing the SPCF based on the short

path activation function only. We will now briefly describe the proposed short-path-based approach to compute the speed-path characteristic function. Consider a gate  $g$  with a single output  $z$  in a technology-mapped circuit with inputs  $a_1, a_2, \dots, a_k$ . Let  $f(a_1, a_2, \dots, a_k)$  denote the Boolean function realized at  $z$ . Let  $\delta_{a_i}$  denote the delay of input  $a_i$  to output  $z$  and  $\Delta_z$  denote the target arrival time at  $z$ . Let  $\overline{SPCF}_z(\Delta_z)$  denote the complement of the SPCF at  $z$ , i.e., it denotes the set of all input patterns such that the value at  $z$  stabilizes before the target arrival time  $\Delta_z$ . Let  $P$  denote the set of all prime implicants in the on-set and off-set of  $f$ . Let  $L$  denote the set of literals in each prime implicant.  $\overline{SPCF}_z(\Delta_z)$  can be expressed as

$$\overline{SPCF}_z(\Delta_z) = \bigvee_{p \in P} \left( \bigwedge_{l \in L} \overline{SPCF}_l(\Delta_z - \delta_l) \right) \quad (4.1)$$

Eqn. 4.1 can be used to recursively compute  $\overline{SPCF}_y$  for each primary output  $y$  of the circuit that contains speed-paths. The runtime for the proposed path-based algorithm is shown in column 6 of Table 4.5. Note that for comparable runtimes with the node-based approach, the proposed algorithm can compute the SPCF exactly. We will now describe a synthesis technique of the error-masking circuit using the SPCF.

In this work, the computation of the SPCF targets all timing paths in the design within 10% and 20% of the critical path delay. Note that approximate logic circuits can mask timing errors arising simultaneously from multiple critical paths. For an output  $y_i$  and a given target delay  $\Delta$ , the SPCF for  $y_i$ ,  $S_i$ , contains inputs that sensitize speed-paths in the fanin cone of output  $y_i$ . Several algorithms have been

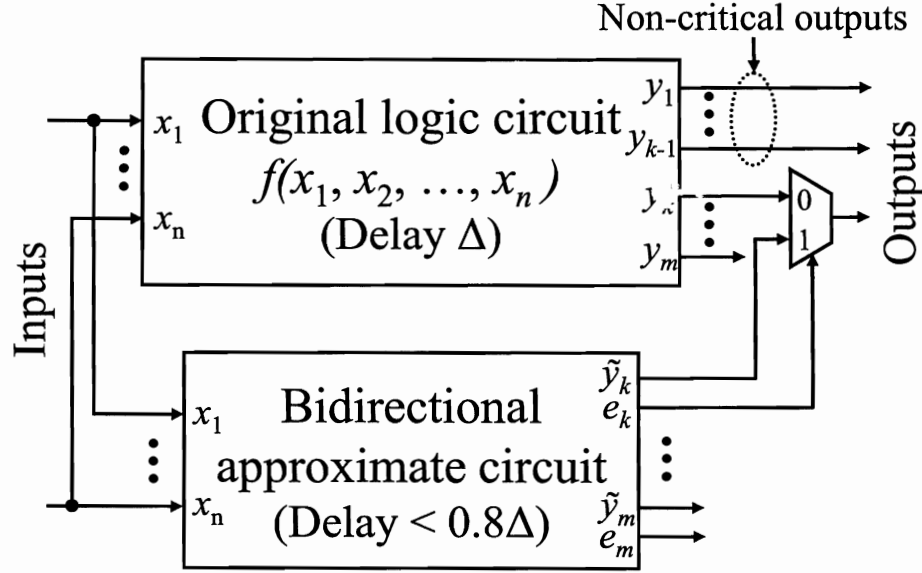


Figure 4.4 : Timing error masking based on bidirectional approximate circuits.

proposed for the exact computation of the SPCF [26; 52]. These algorithms compute the exact set of minterms that sensitize paths with a delay greater than or equal to a desired value. These algorithms are path-based and require traversal of each critical path. Other algorithms that compute an approximation of the SPCF have also been proposed [53; 51]. These algorithms compute an over-approximation of the SPCF, i.e., minterms that do not sensitize critical paths may be included in the SPCF. The over-approximation algorithms are computationally more efficient than path-based algorithms because they are node-based and require computation only at nodes that lie on the critical path. Timing errors can be masked at primary outputs that contain speed-paths, referred to as critical primary outputs, using a bidirectional approximate logic circuit as shown in Fig. 4.4. In the bidirectional approximation of a critical primary output  $y_i$ , the indicator output  $e_i$  is 1 and the predictor output



$\tilde{y}_i$  predicts  $y_i$  correctly for inputs in  $\Sigma_i$ . Error masking at the output is performed using a 2-to-1 multiplexer. Output  $y_i$  is the 0-data input and predictor output  $\tilde{y}_i$  is the 1-data input to the multiplexer. The indicator output,  $e_i$ , is the select input to the multiplexer. When an input applied to the circuit sensitizes a speed-path in the fanin cone of  $y_i$ , i.e., when an input in  $\Sigma_i$  is applied, the select input,  $e_i$ , is 1 and the 1-data input,  $\tilde{y}_i$ , is passed through the multiplexer, thus masking a potential timing error at  $y_i$ . On the other hand, when  $e_i$  is 0, output  $y_i$  is not vulnerable to timing errors and hence the multiplexer passes the output  $y_i$ . Note that the bidirectional approximate circuit is not vulnerable to timing errors since it has at least 20% smaller critical path delay than the given circuit. Hence, unlike errors arising due to latent defects and single-event upsets, timing errors can be masked using a bidirectional approximate circuit. Note that a unidirectional approximation can also be used to mask timing errors. However, with a unidirectional approximation, it will not be possible to achieve 100% masking of timing errors at an output  $y_i$ , if its SPCF,  $S_i$ , contains inputs from both the on-set and the off-set of  $y_i$ .

Timing error masking based on approximate logic circuits has the following advantages over existing approaches:

- Unlike timing error detection techniques [18; 15; 16; 13] that require roll-back or local instruction replay to correct timing errors, timing errors can be masked concurrently during normal operations using approximate logic circuits. Further, resampling based error detection techniques also suffer from

data path metastability and increased clock energy due to the addition of an extra latch [49].

- Unlike error prediction techniques [48; 17] that can predict timing errors resulting from gradual slowdown of speed-paths, e.g., due to aging mechanisms, approximate logic circuits can be used to mask timing errors arising due to gradual slowdown of speed-paths as well as due to fast-changing dynamic variability effects like supply voltage and temperature variations.
- Unlike offline architectural techniques such as periodic stress testing [54], lifetime-reliability tracking based on technology parameters [55], and on-chip temperature and voltage sensors to predict temperature surges and voltage droops [56] that can target only specific sources of timing errors, timing error masking based on approximate circuits is an online technique that can mask errors arising due to a broad range of failure mechanisms.

Simulation results for timing errors on speed-paths arising due to dynamic variability are obtained as follows. Given a mapped logic circuit, dynamic variability can cause timing errors at an output when a speed-path (timing path within 10% or 20% of the critical path delay depending on the extent of dynamic variability) is sensitized. Hence, the input patterns that sensitize the timing paths within 10% or 20% of the critical path delay constitute the specified input sub-space,  $S$ , that is targeted for the approximation. A bidirectional approximate circuit that predicts all the inputs in  $S$  correctly is synthesized using Algorithm 1 and is used to mask potential timing

Table 4.6 : Area and power overhead for 100% concurrent masking of timing errors on speed-paths.

Circuit	Timing paths within 10% of critical path delay					Timing paths within 20% of critical path delay				
	Critical POs	SPCF	Overhead (%)		Slack (%)	Critical POs	SPCF	Overhead (%)		Slack (%)
			Area	Power				Area	Power	
cmb	1/4	$4 \times 10^3$	32	16	52	2/4	$5 \times 10^3$	45	42	34
x2	1/7	16	9.5	3.4	74	2/7	48	38	9.3	46
il	3/16	$5.6 \times 10^6$	40	28	41	3/16	$1.6 \times 10^7$	52	48	32
cu	4/11	$3.6 \times 10^3$	10.4	3.4	77	5/11	$1.6 \times 10^4$	16.6	6	62
cc	6/21	$1.3 \times 10^5$	42	22	30.4	6/21	$4.8 \times 10^4$	65	34	26.2
too_large	2/3	$8.7 \times 10^7$	25	11.5	67	3/3	$1.7 \times 10^9$	85	68	22.3
frg2	12/139	– <sup>†</sup>	–	–	–	36/139	$3.5 \times 10^{18}$	33	11.9	56
i10	3/224	– <sup>†</sup>	–	–	–	9/224	$1.7 \times 10^{69}$	36.5	14.1	41
sparc_ifu_dec	3/146	$4.2 \times 10^{31}$	12.6	3.4	56	15/146	$5.3 \times 10^{37}$	37	18	31
sparc_ifu_dcl	6/94	$7.9 \times 10^5$	15.9	2.8	72	6/94	$1.6 \times 10^{23}$	52.4	39.3	54
lsu_expctl	16/179	– <sup>†</sup>	–	–	–	80/179	$1.8 \times 10^{75}$	24.8	9.8	42
sparc_ifu_errctl	71/399	$7.1 \times 10^{103}$	18.7	5.6	83	71/399	$7.1 \times 10^{103}$	18.7	5.6	83
<b>Average</b>	–	–	22.9	8	61.4	–	–	42	25.5	44.1

<sup>†</sup> The ‘–’ indicates that the speed-paths were not sensitized by any input.

errors in the original logic circuit. A design layout, instead of a mapped logic circuit, can provide more accurate timing information. We believe that our algorithms can be directly applied when an integrated logic and physical synthesis environment is used for extraction of timing-critical computation. Since the focus of this work is the synthesis and application of approximate logic circuits, our simulation results do not use physical timing characteristics.

Table 4.6 presents area and power overhead of the bidirectional approximate circuit used for concurrent masking of timing errors arising on all timing paths within 10% and 20% of the critical path delay. The benchmark circuits considered are shown in the first column. The inputs/outputs, gate count, and area of these benchmark circuits are indicated in Table 4.4. For each benchmark circuit, the number of critical

primary outputs, i.e., the primary outputs containing critical paths and the size of the SPCF are indicated in the first two sub-columns for the top 10% and top 20% critical paths. The next three sub-columns indicate the area, power overhead, and timing slack of the bidirectional approximate circuit used to mask timing errors on the critical paths. The average area (power) overhead of the error-masking circuit is 23% (8%) and 42% (26%) for the top 10% and top 20% critical paths respectively. The average timing slack in the bidirectional approximate circuit over the original circuit was 61% and 44% for the top 10% and top 20% critical paths respectively.

A commonly used technique for achieving timing closure in the presence of dynamic variability is to add timing margins during synthesis and logic optimization. By adding timing margin of say, 10%, the performance of the design is over-optimized by 10%, either using logic optimization, gate sizing, or other technology-dependent circuit tuning techniques. The choice between over-optimizing a design using circuit tuning techniques like gate sizing and using an approximate circuit depends on the logic optimization and gate sizing effort invested into meeting the target performance. Low performance designs usually have a lot of room for improvement in performance using gate sizing and hence, over-optimization of performance using gate sizing is better than using an approximate circuit. An approximate circuit is more useful in high performance designs where performance improvement with gate sizing is either not possible or is more expensive. For these designs, approximate circuits provide a non-intrusive and technology-independent solution for improving performance by

fixing timing-critical paths.

Engineering change order (ECO) techniques such as gate re-sizing and low  $V_T$  swapping are also used to achieve timing closure by fixing timing paths by minimally perturbing the design during later stages of the design cycle. When approximate circuits are used to mask timing errors in a design, then a re-spin of the approximate circuit may be needed when the design is subjected to ECO techniques. For instance, if ECO introduces new timing paths into the set of speed-paths targeted by the approximation, then a new approximate logic circuit targeting the timing-critical input sub-space of the new set of speed-paths has to be synthesized for timing error masking. However, if ECO reduces the set of speed-paths targeted by the approximate circuit, then the approximate circuit can be left unchanged.

**Debug information:** The error-masking circuit can also assist post-silicon at-speed in-system debug by guiding selective capture of debug information in trace buffers. Trace buffers are very useful because they can be used for real-time at-speed observation of limited signals during in-system debug [57; 58]. However, trace buffers can only store a limited amount of data in one debug session. To optimize usage of trace buffers, selective storage of only a few suspect clock cycles has been proposed in [59]. Since errors occur mainly as timing errors on speed-paths [15], and the indicator outputs  $e_i$ s indicate the occurrence of input patterns that sensitize speed-paths, the debug information for only those input patterns may be stored in the trace buffers. Thus, by storing debug information for only vulnerable input patterns, the window

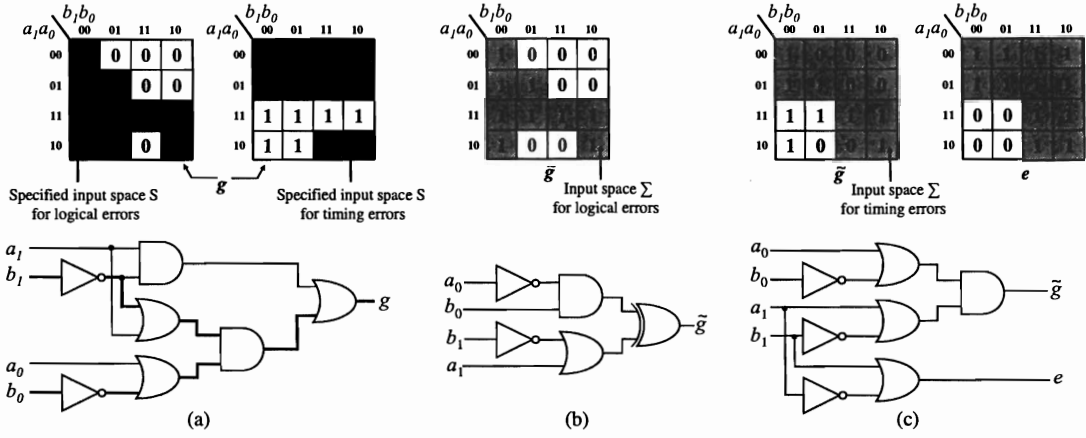


Figure 4.5 : (a) 2-bit comparator with two speed-paths, (b) approximate circuit for masking logical errors, and (c) approximate circuit for masking timing errors.

size of trace buffers can be expanded significantly. In addition, since only one debug session is required, the proposed technique can also be used for debugging unreproducible bugs. In addition to post-silicon debug, the error-masking circuit can also be used during runtime to detect the onset of wearout. As speed-paths slow-down due to wear-out and aging, timing errors will start increasing. With the proposed error-masking circuit in place, these timing errors will be masked. However, the information that a timing error occurred can be recorded and used to detect the onset of wearout of speed-paths.

### 4.3 Example: 2-bit comparator

We will now illustrate concurrent error masking based on approximate circuits for a 2-bit comparator. A 2-bit comparator compares two 2-bit binary number,  $a_1a_0$  and  $b_1b_0$ . The output,  $y$ , of a 2-bit comparator is 0 when the decimal equivalent of  $a_1a_0$

is less than the decimal equivalent of  $b_1b_0$ . The K-map for  $y$  and a delay-optimized circuit implementation of a 2-bit comparator are shown in Figure 4.5(a). The shaded cells in the K-map of  $y$  indicate the specified input space,  $S$ , that are vulnerable to errors. The K-map on the left shows the input space vulnerable to errors arising due to latent defects and single-event upsets. Note that since the on-set of  $y$  is larger than its off-set, we can maximize error masking coverage by masking  $1 \rightarrow 0$  errors at  $y$ . Hence,  $S$  is the on-set of  $y$ . Based on this, Figure 4.5(b) shows the unidirectional approximate logic circuit that predicts 87.5% of the specified input space,  $\Sigma$ .

The K-map on the right shows the input space vulnerable to timing errors on speed-paths. The speed-paths are highlighted in the circuit diagram of the 2-bit comparator. Fig. 4.5(c) shows a bidirectional approximate circuit that predicts 100% of the specified input space  $S$  correctly. Note that the approximate logic circuit is not susceptible to timing errors since it has a smaller delay than the 2-bit comparator.

## Chapter 5

### Time borrowing and error relaying (TIMBER)

Timing errors in a microprocessor can be masked by synthesizing an approximate logic circuit (as described in Chapter 4) for every output and flip-flop that contains critical paths. Although this approach eliminates the performance overhead incurred for roll-back in timing error detection approaches [15], the combinational logic overhead may be significant, especially for high-performance microprocessors that have a large number of flip-flops and outputs containing critical paths. The time borrowing and error relaying (TIMBER) architecture was developed with the goal of reducing combinational logic overhead required for timing error masking using approximate circuits.

Timing analysis of critical paths in an ARM processor showed that only a small fraction of flip-flops serve as *both* start and end-points of critical paths. Hence, timing errors occurring at a significant fraction of flip-flops can be masked by borrowing time from successive pipeline stages. The TIMBER architecture proposes new flip-flop/latch designs that are capable of masking timing errors occurring in a pipeline stage by borrowing from the successive pipeline stage. Thus, timing errors arising from local dynamic variations and fast changing global dynamic variations that frequently span only a single pipeline stage can be masked using TIMBER. Further,



TIMBER proposes architectural modifications to mask timing errors spanning multiple pipeline stages, e.g., due to slow changing global variations. A multi-stage timing error occurs when two or more critical paths are affected by dynamic variability on successive clock cycles across multiple stages. In such cases, TIMBER enables the system to run error-free for multiple cycles during which it initiates a temporary reduction of the clock frequency at the system level to mitigate the occurrence of timing errors. Hence, multi-stage timing errors can be masked without requiring hardware support for roll-back or instruction replay. Since the probability of a multi-stage timing error is very small, the loss in performance due to a temporary reduction in clock frequency is negligible.

## 5.1 Related work and its limitations

Broadly, existing techniques can be classified into three categories — error detection, error prediction, and error masking.

**Error detection:** Error detection techniques are based on monitoring data-path signals for transitions arriving *after* the clock edge. In [15], one of the earliest circuits for timing error detection using an online stability checker that monitored late transitions arriving in a stability checking period after the clock edge was described. In [16; 60], a sensing circuit for delay faults for self-checking applications was described. In [18], error detection based on re-sampling data-path signals after a delay, and then comparing the resampled value to the value stored in the data-path flip-flop was proposed.

RAZOR [19] proposed the application of this online timing error detection scheme to reduce power or increase performance using runtime voltage/frequency tuning. A variant of RAZOR that replaces the data-path flip-flop by a latch to avoid metastability issues was proposed in [21]. However, the duty cycle of the clock has to be adjusted to avoid severe hold-time constraints introduced by the latch. In [13], an error detection circuit based on a sense amplifier that can detect both timing errors and soft errors was described. Logic-based techniques for concurrent delay testing (e.g., [61]) based on circuit duplication have also been proposed.

**Error prediction:** Error prediction techniques are based on monitoring data-path signals for transitions for a specified time period *before* the clock edge. In [17], a stability checker design that predicts timing errors due to a gradual increase in delay due to wearout and aging effects was described. Another error prediction technique that pads the data-path with a delay element and samples the delayed data-path signal in another flip-flop, called the canary flip-flop, was described in [20]. A timing error is predicted when the value in the data-path flip-flop differs from the value in the canary flip-flop. Error prediction based on duplicating critical paths and using timing errors on the duplicated paths to predict a timing error on the original paths was described in [50]. This approach is limited in its effectiveness since (i) the duplicated and critical paths in the design may experience different workloads and variability and (ii) the critical paths may change over time [62].

**Error masking:** Error masking techniques proposed in literature can be classified

into two categories: logical and temporal. Logical error masking techniques (e.g., [63]) use redundant logic to compute the correct value of the output with a smaller delay when critical paths are exercised. Temporal error masking techniques mask errors by time-borrowing, i.e., delaying the arrival time of the correct data to the next pipeline stage. In [64], a temporal error masking technique based on stalling the clock for one cycle after detecting a timing error to correct the state of the system was proposed. This technique assumes that the latency for consolidating errors from various flip-flops in the design is less than a clock cycle to stall the clock before the state is corrupted. However, in practice, this may be difficult to achieve in high performance designs due to (i) a small cycle time and (ii) long latency involved in consolidating error signals from a large number of flip-flops susceptible to timing errors. In [65], an edge detector detects timing violations near the clock edge, and a delayed clock is subsequently used to resample and correct the data-path value by borrowing time from the next pipeline stage. This technique assumes that the time borrowed from the next pipeline stage is absorbed by a non-critical path being sensitized in the next stage. This may not be a valid assumption and may lead to timing errors, especially in high performance designs. Further, the edge detector circuit depends on accurate delay values and margining may be needed in the presence of process variations. Finally, a mathematical formulation for time-borrowing in a linear pipeline using a soft-edge flip-flop was described in [66].

A comparison of TIMBER architecture to several techniques for improving relia-

bility proposed in literature is presented in Table 5.1.

Table 5.1 : Comparison of various techniques for online timing error resilience.

Feature	Error detection	Error prediction	Error masking	
			Logical	Temporal
Error detection mechanism	Duplicate latch/FFs Transition detectors	Duplicate latch/FFs Sensors Duplicate paths	Redundant logic	Duplicate latch/FFs Edge detectors
When? (Relative to clock edge)	After	Before	–	After
Error recovery mechanism	Rollback	No error	No error	No error
Clock-tree loading	Yes	Yes	No	Yes
Short-path padding	Yes	Yes	No	Yes
Sequential overhead	Large	Large	None	Large
Combinational overhead	Small	None	Moderate	Small
Timing margin recovery	Full	Partial	Full	Full
Variability source targeted	All dynamic	Gradual dynamic	All dynamic	All dynamic
Techniques	RAZOR [19] TDTB [21] DSTB [21]	Canary FFs [20] Sensors [17] TRC [50]	Approximate circuits	PEDFF [64] DCFF [65] <b>TIMBER</b>

## 5.2 TIMBER: Overview

The distribution of critical paths in an industrial processor shows that timing errors caused by dynamic variations frequently span only a single pipeline stage on successive clock cycles, and thus can be motivates the potential for error masking based on time-borrowing. For a critical path  $p$ , a single-stage timing error is defined as the event that dynamic variability causes the delay of  $p$  to exceed the clock period. Consider multiple critical paths,  $p_1, p_2, \dots, p_k$  such that the terminal flip-flop of  $p_{i-1}$  is the starting flip-flop of  $p_i$ ,  $1 < i \leq k$ . For  $k > 1$ , a  $k$ -stage (i.e., multi-stage) timing error is defined as the event that over  $k$  successive clock cycles, dynamic variability causes a  $(k-1)$ -stage timing error on paths  $p_1, \dots, p_{k-1}$  and the sum of the delays on  $p_1, \dots, p_k$  exceeds  $k$  times the clock period.

Fig. 5.1(a) summarizes the critical path distribution between flip-flops in an industrial processor. Three performance points — low, medium, and high — were considered. There are four bars for each performance point corresponding to the percentage of flip-flops that have a path in the top 10%, 20%, 30%, and 40% critical paths terminating at them. The shaded portion of each bar indicates the percentage of flip-flops that have critical paths starting and terminating at them. Consider the top 20% paths in the medium performance processor. Although nearly 50% of the flip-flops have critical paths terminating at them, 70% of these flip-flops do not have any top 20% critical path originating from them in the next pipeline stage. Hence, 70% of the flip-flops have at least 20% timing slack on all paths in the successive pipeline stage, and are only susceptible to single-stage timing errors.

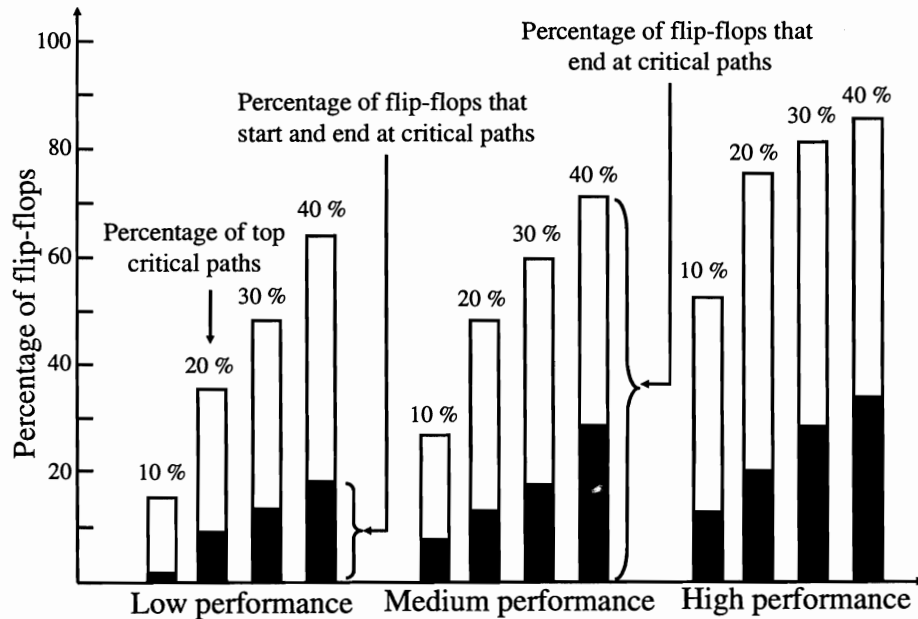


Figure 5.1 : Critical path distribution between flip-flops.

The remaining 30% flip-flops have critical paths starting and terminating at them, and thus are susceptible to multi-stage timing errors. For a multi-stage timing error to occur, multiple critical paths connected end-to-end will have to be sensitized on successive clock cycles. The critical path sensitization probability for the top 10% critical paths is of the order of  $10^{-4}$ – $10^{-8}$  [63]. Hence, the probability of a multi-stage timing error resulting from sensitization of multiple critical paths on successive clock cycles is negligibly small. To summarize, the single-stage timing error rate due to dynamic variability effects is much higher than the multi-stage timing error rate in modern processors. In TIMBER, multi-stage timing errors are masked by time-borrowing up to two or three stages. Multi-stage timing errors spanning more stages are avoided by reducing the clock frequency at the system level. Based on the latency incurred for error consolidation, the clock frequency can either be reduced immediately after the first timing error occurs or it can be deferred until the first multi-stage timing error occurs. Both approaches have their merits and the trade-offs are discussed in detail in the next section.

The core principle of TIMBER is to detect a timing error after the clock edge and to mask the timing error by borrowing time from the next pipeline stage. By masking timing errors, TIMBER can recover timing margins required to offset dynamic variability effects without roll-back or instruction replay. The largest timing violation that can occur due to dynamic variability effects in a single pipeline stage is equal to the timing margin that we seek to recover using TIMBER, henceforth termed the

recovered timing margin. The period of time after the clock edge reserved for error detection and masking is referred to as the checking period. The duration of the checking period and the recovered timing margin are fixed during design, and they are related as follows. When the first timing error, i.e., a single-stage timing error occurs, the late arriving data signal can cause a worst-case timing violation equal to the recovered timing margin. TIMBER can mask this single-stage timing error by borrowing a time interval of duration equal to the recovered timing margin. If the next pipeline stage is also affected by dynamic variability, then the late arriving data signal can cause a worst-case timing violation equal to twice the recovered timing margin. TIMBER can mask a two-stage timing error by borrowing a time interval of duration equal to twice the recovered timing margin. In general, for a given checking period,  $c$ , and a recovered timing margin,  $t$ , TIMBER can mask up to  $k$ -stage timing errors such that  $c = k \times t$ . Thus, the checking period can be divided into  $k$  intervals, each of duration  $t$ . Note that the checking period determines the hold-time constraints for the design, i.e., the short paths in the design must be padded to have a delay greater than the sum of hold time and the checking period.

The time intervals in the checking period are classified into two types: time-borrowing (TB) and error-detection (ED), such that the first  $k_{\text{TB}} \geq 0$  intervals are of type TB and the rest of the  $k_{\text{ED}} = k - k_{\text{TB}}$  intervals are of type ED. TIMBER is designed to mask up to  $k$ -stage timing errors, and to avoid a  $(k+1)$ -stage timing error by reducing the clock frequency at the system level. Timing errors up to  $k_{\text{TB}}$

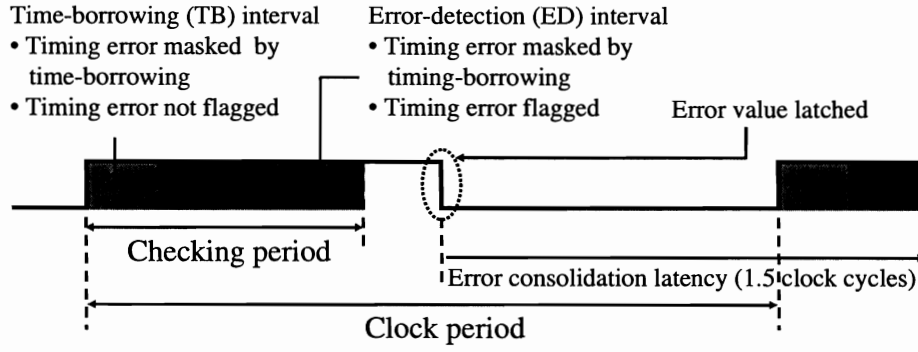


Figure 5.2 : TIMBER-based error detection and error masking.

stages can be masked by borrowing only the TB time intervals in the checking period. These timing errors are not flagged to the central error control unit, i.e., flagging of timing errors to the central error control unit are deferred to the first occurrence of a  $(k_{TB}+1)$ -stage timing error. The advantage of deferring error flagging is that timing errors can be masked without reducing the clock frequency. However, the TB intervals required for deferring error flagging result in lesser recovered timing margin since the checking period has to be divided into more intervals of shorter duration. Hence, the recovered timing margin can be increased for the same checking period by eliminating the TB interval, i.e., by flagging single-stage timing errors to the central error control unit.

On the other hand, masking a  $(k_{TB}+1)$ -stage timing error requires borrowing an ED interval (in addition to all  $k_{TB}$  time intervals) in the checking period. A  $(k_{TB}+1)$ -stage timing error is the first timing error that is flagged to the central error control unit. The remaining  $(k_{ED}-1)$  ED intervals ensure that all timing errors are masked for  $(k_{ED}-1)$  clock cycles after the first timing error is flagged to the central



error control unit. The error signal is latched on the falling edge of the clock cycle on which the first ED time interval is borrowed, i.e., when the first  $(k_{\text{TB}} + 1)$ -stage timing error occurs. This provides an extra half clock cycle for error consolidation. The error signals from all TIMBER sequential elements is consolidated using an OR-tree at central error control unit. After an error consolidation latency, attributed mainly to the latency of the OR-tree, the central error control unit reduces the clock frequency temporarily at the system level to mitigate the timing error rate.

Fig. 5.2 shows a checking period that is divided into three intervals, with one TB interval and two ED intervals. Based on this, all single-stage timing errors are masked but not flagged to the central error control unit. Two-stage timing errors are masked by borrowing a TB and an ED time interval and the timing error is also flagged to the central error control unit. Since there are two ED intervals, the second ED interval ensures that all timing errors are masked for  $(k_{\text{ED}} - 1)$  clock cycles after the first timing error is flagged to the central error control unit. Hence, the error consolidation latency must be less than 1.5 clock cycles (half a clock cycle is added because the error signal is latched on the falling clock edge).

The next section describes two types of sequential elements are proposed to implement time-borrowing in the TIMBER architecture: TIMBER flip-flop and TIMBER latch. TIMBER flip-flop implements time-borrowing in discrete units, thus preserving the edge-sampling property of a conventional master-slave flip-flop. On the first pipeline stage with a timing error, TIMBER flip-flop masks the error by borrowing

one time unit. An error relay logic alerts the next stage to borrow an additional time unit, if a timing error propagates to that stage. On the other hand, TIMBER latch implements continuous time-borrowing, i.e., TIMBER latch is transparent for the entire checking period (equal to multiple discrete time units), and hence any late arriving transition in the checking period is masked by borrowing time. Thus, TIMBER latch does not require error relay logic. Although the edge-sampling property of TIMBER flip-flop is lost and TIMBER latch propagates glitches and spurious transitions in the checking period, our implementation guarantees that TIMBER latch does not signal a false timing error.

### 5.3 TIMBER: Circuit design

This section describes time-borrowing sequential circuit designs for the TIMBER architecture. Two time-borrowing flip-flop designs are proposed: (i) TIMBER flip-flop uses three latches — two master latches and one slave latch — to implement discrete time-borrowing. This design provides flexibility for dynamic configuration of a TIMBER flip-flop as a conventional master-slave flip-flop (without any time-borrowing) and (ii) dedicated TIMBER flip-flop uses only two latches to implement discrete time-borrowing, but sacrifices the flexibility for dynamic configuration as a conventional master-slave flip-flop. Both TIMBER flip-flops preserves the edge-sampling property of a master-slave flip-flop because error masking is performed by borrowing discrete time intervals. As a result, when these TIMBER flip-flops are

used, the TIMBER architecture requires error relay logic to determine the number of time intervals required to mask errors spanning multiple pipeline stages. The TIMBER latch design proposed in this section uses only two latches to implement continuous time-borrowing and provides flexibility for dynamic configuration of a TIMBER latch as a conventional master-slave flip-flop. TIMBER latch eliminates the need for error relay logic by implementing time-borrowing using a level-sampling latch. However, TIMBER latch propagates glitches and spurious transitions during the checking period. The TIMBER architecture described in this section divides the checking period into one TB and two ED intervals.

### 5.3.1 TIMBER flip-flop

A TIMBER flip-flop consists of two master latches, M0 and M1, and a common slave latch as shown in Fig. 5.3(a). The clock control logic for the TIMBER flip-flop is shown in Fig. 5.3(b). The signal R denotes the system reset signal and the signal EN is the enable signal. Time-borrowing in a TIMBER flip-flop can be turned off by setting EN to zero. When EN is low, P0 is  $\overline{CK}$ , and P1 is high. Thus, M0 and the slave latch together function as a conventional master-slave flip-flop and M1 is blocked because the transmission gate P1 is open. In a conventional master-slave flip-flop, M0 samples the value of the data signal D at the rising edge of the CK and drives the slave latch and the output Q to the sampled value when CK is high. When CK goes low, the transmission gate P0 is open and the slave latch drives the output

Q.

When EN is high, the TIMBER flip-flop operates in the time-borrowing mode. The three intervals in the checking period are encoded using the select input signals,  $S_1S_0$ .  $S_1S_0 = 00$  is the TB interval and  $S_1S_0 = 01, 10$  are the ED intervals. On system reset,  $S_1S_0$  is set to 00. Error masking based on time-borrowing happens as follows. The master latch M0 samples the value of the data signal, D, on the rising edge of clock and drives the slave latch and the output, Q, to the sampled value. The master latch M1 samples the data signal, D, on the rising edge of the delayed clock, DCK, after a delay  $\delta$  determined by the value of the select inputs  $S_1S_0$ . On the rising edge of the delayed clock, DCK, the transmission gate P0 opens and the transmission gate P1 closes. Thus, after delay  $\delta$ , for the rest of the clock period when CK is high, the master latch M1 drives the slave latch and the output Q to the new value sampled by M1. If no timing error has occurred, the master latches M0 and M1 would sample the same value. Hence, M0 drives the slave latch and the output to the correct value on the rising edge of CK, and no time-borrowing occurs.

If a timing error occurs at the flip-flop, the master latches M0 and M1 sample different values, and M1 masks the timing error after delay  $\delta$  as follows. Recall that error masking in a TIMBER flip-flop occurs by borrowing discrete time units. Suppose each interval in the checking period has a duration of 100ps, and  $S_1S_0$  is 00. If a timing error occurs due to a 80ps timing violation on the data input, then the error is masked by the master latch M1 after a 100ps delay, i.e., 100ps is borrowed

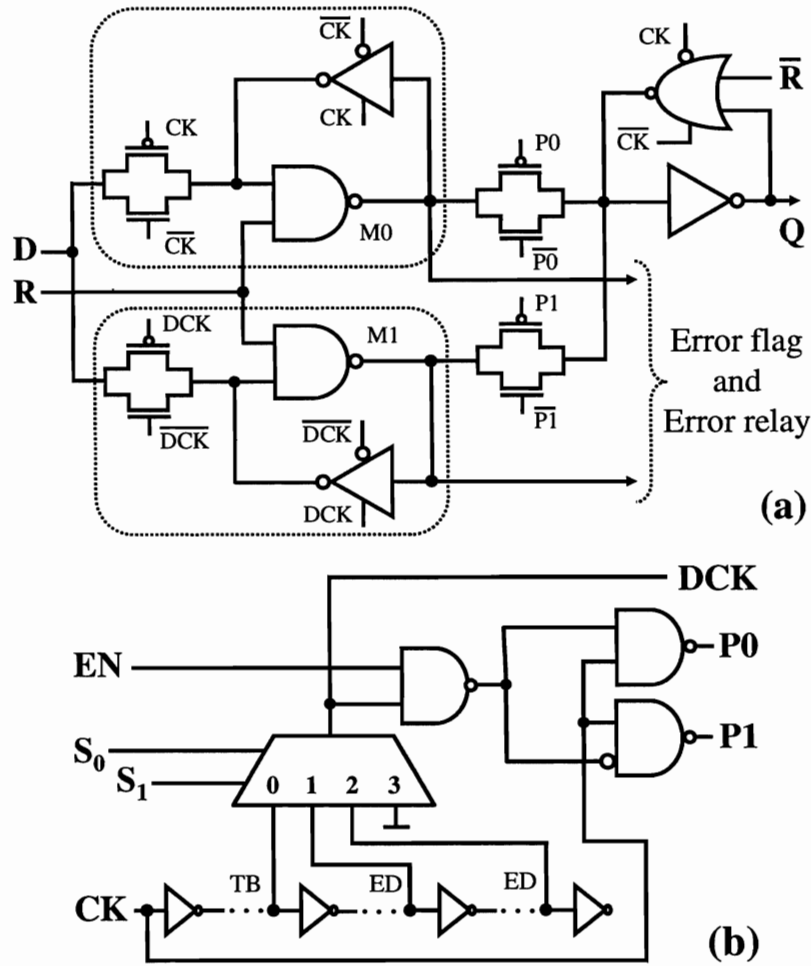


Figure 5.3 : TIMBER flip-flop (a) design and (b) clock control.

from the next stage. Note that TIMBER flip-flop does not suffer from data-path metastability issues because a data-path signal violating setup time on the rising edge of clock is masked by the delayed sampling of the data-path signal by master latch M1. To mask multi-stage timing errors, error relay logic configures the select inputs of TIMBER flip-flops in successive pipeline stages as follows.

**Error relay:** Consider a TIMBER flip-flop,  $f$ , with  $m$  TIMBER flip-flops  $g_1, g_2, \dots, g_m$

in the fanin cone of  $f$ . Denote  $S(g_i)$  as the select input to  $g_i$ . If no error occurs at  $g_i$ , then the select output of  $g_i$  is set to 00. If an error occurs at  $g_i$ , then the select input  $S(g_i)$  is incremented by 1 to obtain the select output for  $g_i$ . Incrementing  $S(g_i)$  by 1 ensures that the TIMBER flip-flop  $f$  can borrow an additional time interval if a multi-stage timing error occurs at  $f$ . The select input for  $f$  is obtained as the maximum over all the select outputs from  $g_1, g_2, \dots, g_m$ . The logic for generating the select outputs at each TIMBER flip-flop using its select inputs is omitted from Fig. 5.3(a) due to space constraints. Fig. 5.4 is the block diagram for the error relay logic. Note that the error relay logic is different from the error consolidation logic to the central error control unit. Recall that the error signal is latched on the falling edge of the clock. Since the error relay logic must set the select inputs before the next rising clock edge, the error relay logic can have a maximum delay of half of the clock period. In Sec. 7.4, a case-study for an industrial processor shows that the delay of the error relay logic is much smaller than half a clock period. This is because the error relay for a TIMBER flip-flop must occur only from a small number of TIMBER flip-flops in its fanin cone that are both start and endpoints of critical paths (refer Fig. 5.1).

Fig. 5.5 shows SPICE waveforms for error masking when a two-stage timing error occurs on two TIMBER flip-flops,  $f_1$  and  $f_2$ , on successive pipeline stages. The signals  $D_1$  ( $D_2$ ),  $Q_1$  ( $Q_2$ ), and  $Err_1$  ( $Err_2$ ) are the data, output, and error signals for flip-flop  $f_1$  ( $f_2$ ). The first timing error, occurring at flip-flop  $f_1$ , is masked by borrowing one

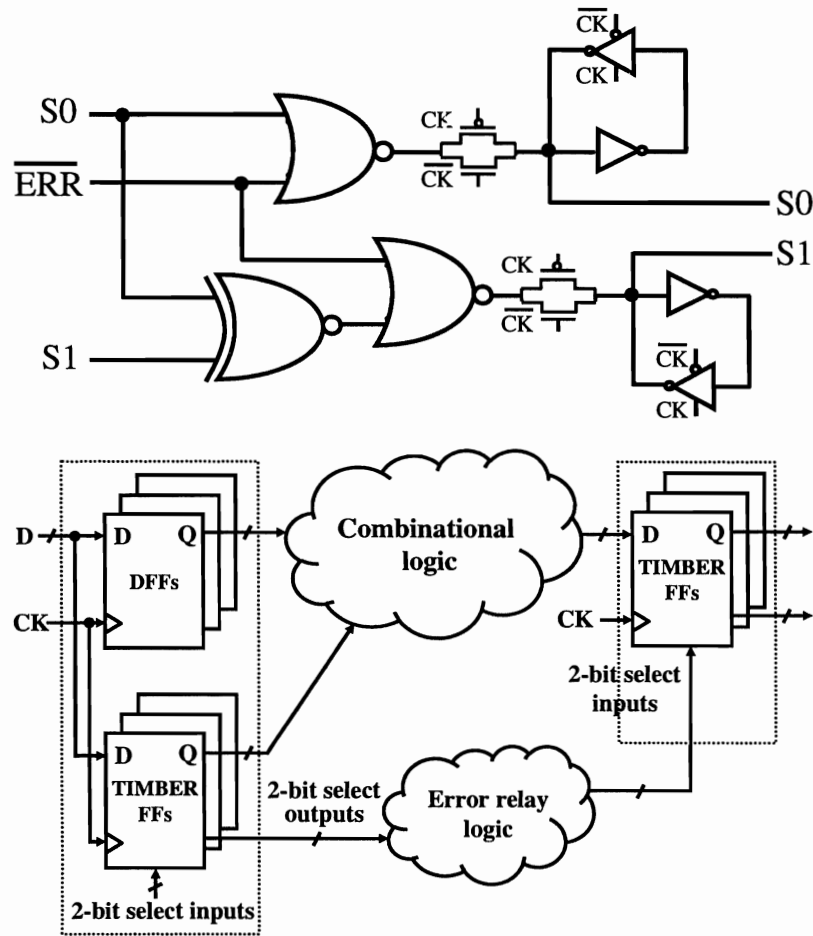


Figure 5.4 : TIMBER flip-flop error relay logic.

TB time interval at  $f_1$ . Although the timing error is not flagged to the central error control unit ( $\text{Err}_1$  signal is 0), the error relay logic configures the select inputs of flip-flop  $f_2$  to 01. Thus, when a two-stage timing error occurs at flip-flop  $f_2$ , the error is masked by borrowing a TB and an ED time interval at  $f_2$ . The timing error at  $f_2$  is flagged to the central error control unit by latching the error signal ( $\text{Err}_2$  signal goes high) on the subsequent falling edge of CK.

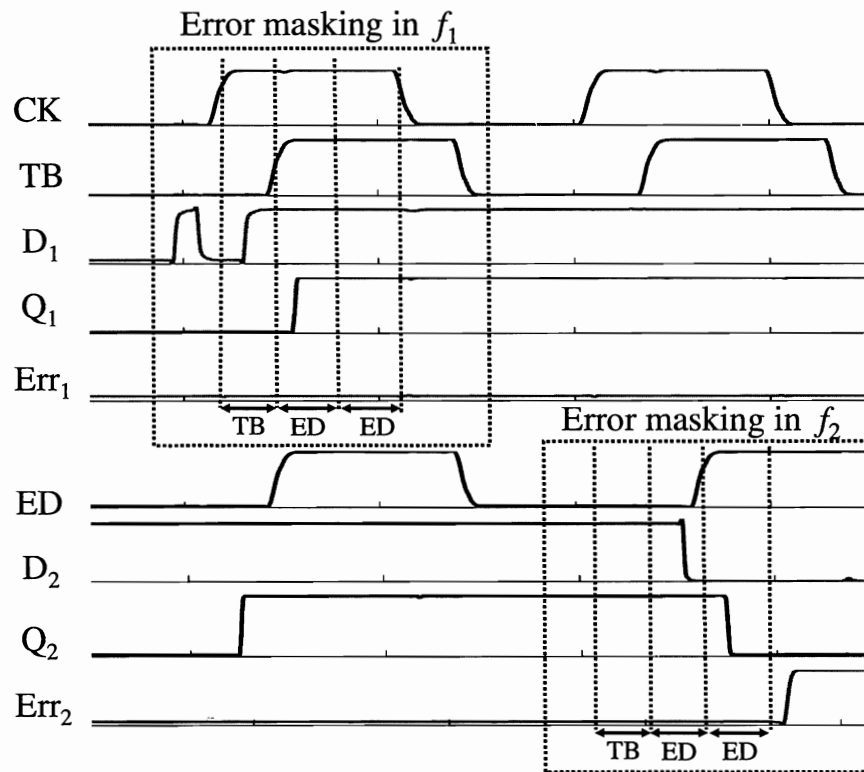


Figure 5.5 : Two-stage timing error in a TIMBER flip-flop design.

### 5.3.2 Dedicated TIMBER flip-flop

A TIMBER flip-flop requires three latches — two master latches and one slave latch. In a conventional master-slave flip-flop, the purpose of the slave latch is to hold the previous data to drive the inputs to the subsequent pipeline stage while the master latch samples the new data. The slave latch in a TIMBER flip-flop serves the same purpose when the TIMBER flip-flop is configured in the non time-borrowing mode. However, when a TIMBER flip-flop is configured in the time-borrowing mode, the slave latch is not required to drive the inputs of the subsequent pipeline stage because the two master latches can switch between sampling data and driving the inputs to



the subsequent pipeline stage. Thus, if a TIMBER flip-flop is required only in the time borrowing mode (dedicated TIMBER flip-flop), then the circuit design of the TIMBER flip-flop can be optimized by eliminating the slave latch (see Fig. 5.6). The master latch M0 of a dedicated TIMBER flip-flop is open when CK is low and samples the data D on the rising of CK. When CK is high and DCK is low, master latch M0 drives the inputs of the subsequent pipeline stage. During this period master latch M1 is open and sample the data D on the rising edge of DCK. Outside this time interval, the master latch M1 drives the inputs of the subsequent pipeline stage. A timing error is detected by comparing the values sampled by the master latches M0 and M1 on the falling edge of CK. The timing diagram of a dedicated TIMBER flip-flop is shown in Fig. 5.7.

### 5.3.3 TIMBER latch

TIMBER latch implements time-borrowing in continuous units using a level-based sampling of the data using a pulse-gated latch. A TIMBER latch consists of a master and a slave latch as shown in the circuit schematic in Fig. 5.8(a). The clock control logic for a TIMBER latch is shown in Fig. 5.8(b). The signal R denotes the system reset signal and the signal EN is the enable signal. Time-borrowing in a TIMBER latch can be turned off by setting EN to zero. When EN is low, the transmission gate L is open and the TIMBER latch operates as a conventional master-slave flip-flop.

When EN is high, the TIMBER latch operates in the time-borrowing mode. In

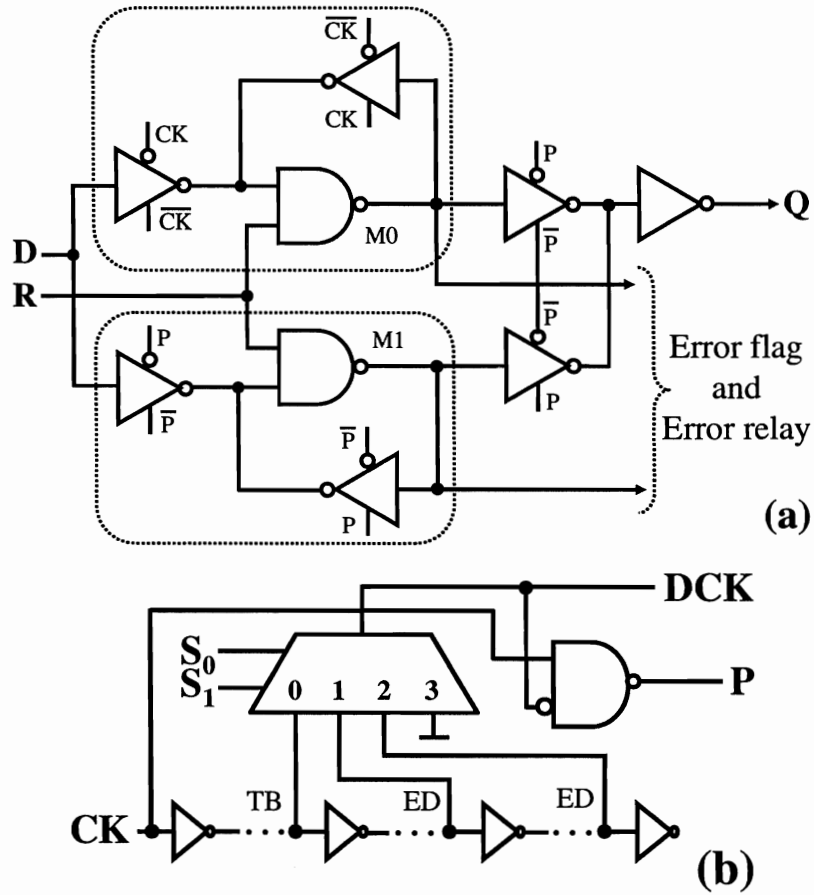


Figure 5.6 : Dedicated TIMBER flip-flop (a) design and (b) clock control.

this mode, the transmission gate F is open and the master latch and slave latch operate independently as pulse-gated latches. The checking period is divided into one TB and one ED interval. Note that this ED interval is equivalent to the sum of the ED intervals in the TIMBER flip-flop. The master latch is transparent during the TB interval and the slave latch is transparent for the entire checking period. A timing error is detected by comparing the values stored in the master latch and the slave latch on the falling edge of the clock. When a single-stage timing error occurs, the

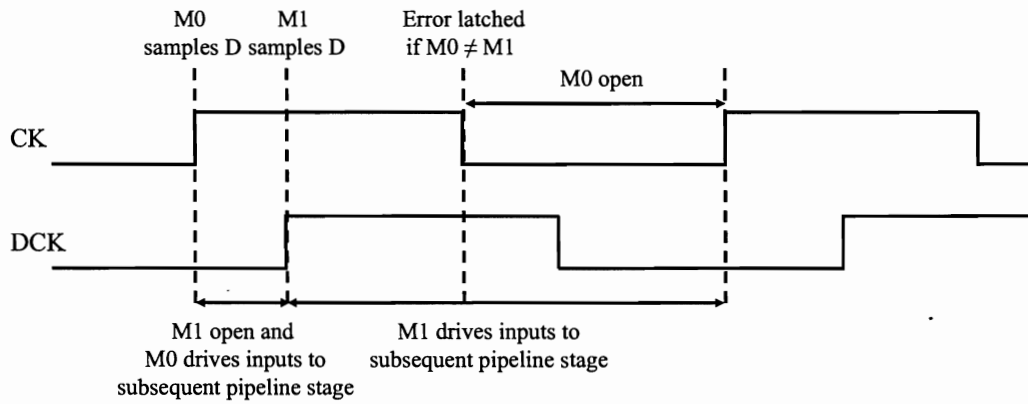


Figure 5.7 : Timing diagram for dedicated TIMBER flip-flop.

timing violation of the late arriving data signal lies within the TB time interval. The timing error is masked because the slave latch is transparent for the entire checking period. Since the master is also transparent for the TB interval, both the master latch and slave latch hold the same value and hence, a timing error is not flagged. However, if a two-stage timing error occurs such that the timing violation of the late arriving data signal is greater than the TB interval, then the master and slave latches sample different values, and a timing error is detected and flagged to the central error control unit. Recall that a TIMBER latch masks timing errors by borrowing continuous time units. Suppose the TB interval is 100ps and a timing violation of 80ps occurs at a TIMBER latch, then the error is masked by borrowing 80ps from the next stage. Since the slave latch is transparent for the entire checking period, error relay logic is not required. However, TIMBER latch propagates glitches and spurious transitions during the checking period. Note that TIMBER latch does not have metastability issues because level-sensitive sampling is used for time-borrowing.



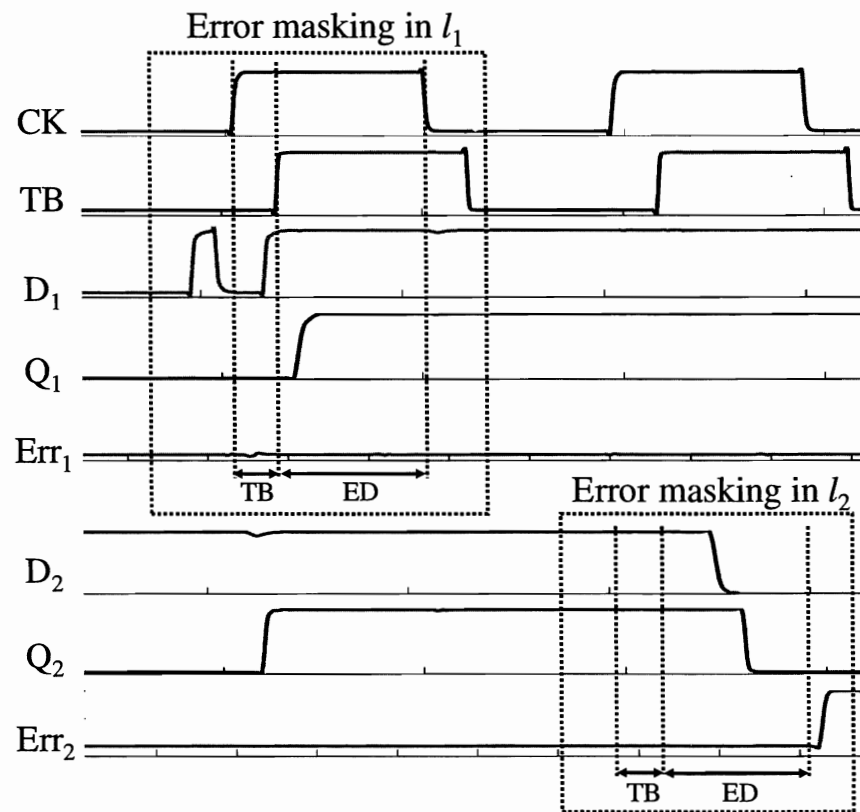


Figure 5.9 : Two-stage timing error in a TIMBER latch design.

signal (Err<sub>2</sub> signal goes high) on the subsequent falling edge of clock CK.

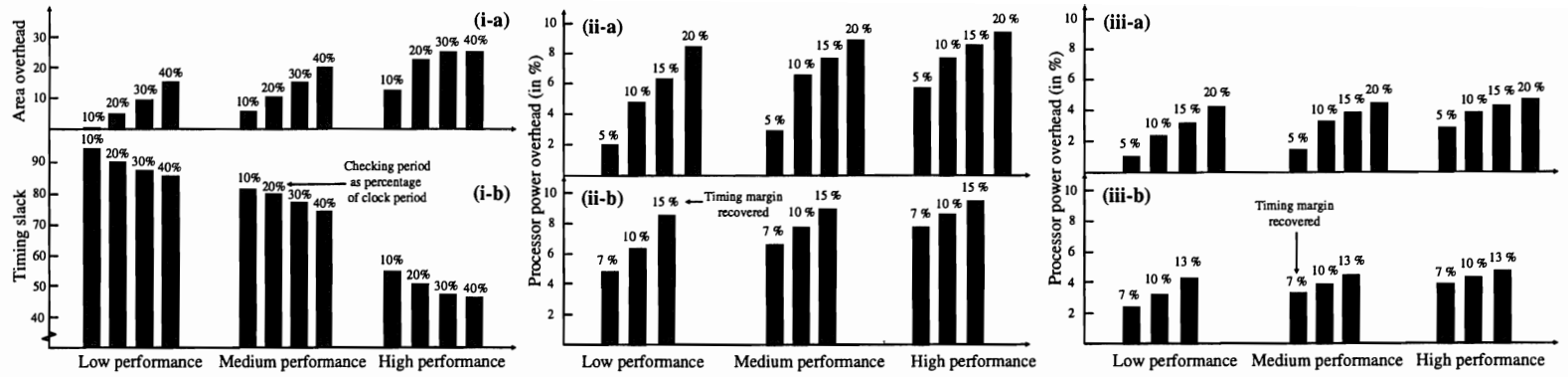


Figure 5.10 : (i) TIMBER flip-flop error relay logic: (a) area overhead and (b) timing slack, (ii) Power overhead for TIMBER flip-flop: (a) without TB and (b) with TB interval, and (iii) Power overhead for TIMBER latch: (a) without TB and (b) with TB interval.

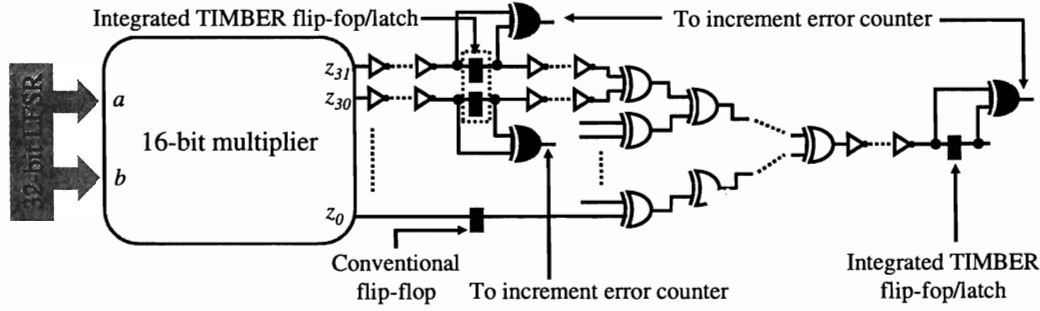


Figure 5.11 : Hardware setup for measuring error rate with and without TIMBER.

## 5.4 Hardware validation

This section describes the test structures implemented on a Nexys2 development board with Xilinx Spartan 3E FPGA. The goal is to validate timing error masking based on TIMBER flip-flop and TIMBER latch by comparing timing error rates with and without TIMBER. The timing error rates are injected in a two-stage pipeline by varying the frequency of operation. The Nexys2 development board provides a 50MHz oscillator for clocking designs in the FPGA. In addition, digital clock management (DCM) capability allows multiplying or dividing the clock frequency by integral factors between 2 and 32, thus providing frequency scaling capability from 5MHz to 311MHz. We have found that frequencies between 50MHz and 100MHz can be generated with at the finest granularity steps and hence our designs are tuned to adhere to this frequency range.

**Test structure:** The test structure consists of three main stages:

- A 32-bit linear feedback shift register (LFSR) that generates pseudo-random input patterns to the first pipeline stage.

- A two stage pipeline for which timing error rate measurements will be performed. The first stage of the pipeline is a 16-bit multiplier with 32 inputs (two 16-bit binary numbers) and 32 outputs. The second pipeline stage uses the latched outputs of multiplier as inputs to a 32-input XOR tree.
- Control logic circuitry for comparing single cycle computation to multi-cycle computation and tracking error rates at various latches/flip-flops using 32-bit synchronous counters.

Fig. 5.11 shows the test structure and Fig. 5.14 shows the timing diagram for measuring timing error rates with and without TIMBER. The two most significant bits of the multiplier outputs ( $z_{31}$  and  $z_{30}$ ) and the output of the 32-bit XOR tree are data signals to the flip-flops/latches for which timing errors will be measured. We pad the timing paths through these outputs by adding inverter chains so that the first timing error at these data signals are observed at 60MHz without TIMBER.

Our goal is to measure timing error rates in two scenarios: with a conventional flip-flop without time-borrowing and with TIMBER flip-flop/latches. For this purpose, we have designed circuits for two sequential elements that integrates a flip-flop without time-borrowing with TIMBER flip-flop (Fig. 5.12) and with TIMBER latch (Fig. 5.13). Integrating sequential elements with and without time-borrowing into a single block has two advantages: (i) the error rates with and without time-borrowing can be measured in a single FPGA run and (ii) eliminates routing and placement differences introduced when the designs are compiled and executed sepa-



rately with and without time-borrowing. Further, clock skew difference is mitigated in an integrated design because the clock tree is shared between flip-flops with and without time-borrowing.

**Integrated TIMBER flip-flop:** The integrated TIMBER flip-flop consists of the four types of flip-flops.

1. A simple flip-flop without any time-borrowing.
2. A TIMBER flip-flop with time-borrowing, but no error relay that borrows a fixed time interval (two inverter delays) in each pipeline stage. Each gate in Fig. 5.12 is implemented using a lookup table (LUT) in an FPGA and thus, two LUT delay time interval is borrowed at each pipeline stage. The purpose of this flip-flop is to demonstrate the importance of error relay in masking multi-stage timing errors.
3. A TIMBER flip-flop with error relay borrows a time interval of two LUT delays (one inverter and one 2-to-1 multiplexer) if no timing error has occurred in the previous stage ( $Err_{i-1} = 0$ ) and borrows a timing interval of four LUT delays (three inverters and one 2-to-1 multiplexer) if a timing error occurs and is masked by time-borrowing in the previous stage ( $Err_{i-1} = 1$ ).
4. A dedicated TIMBER flip-flop also borrows variable time intervals based on the error relay from the previous pipeline stage. A dedicated TIMBER flip-flop can only operate in the time-borrowing mode and this helps eliminate the slave

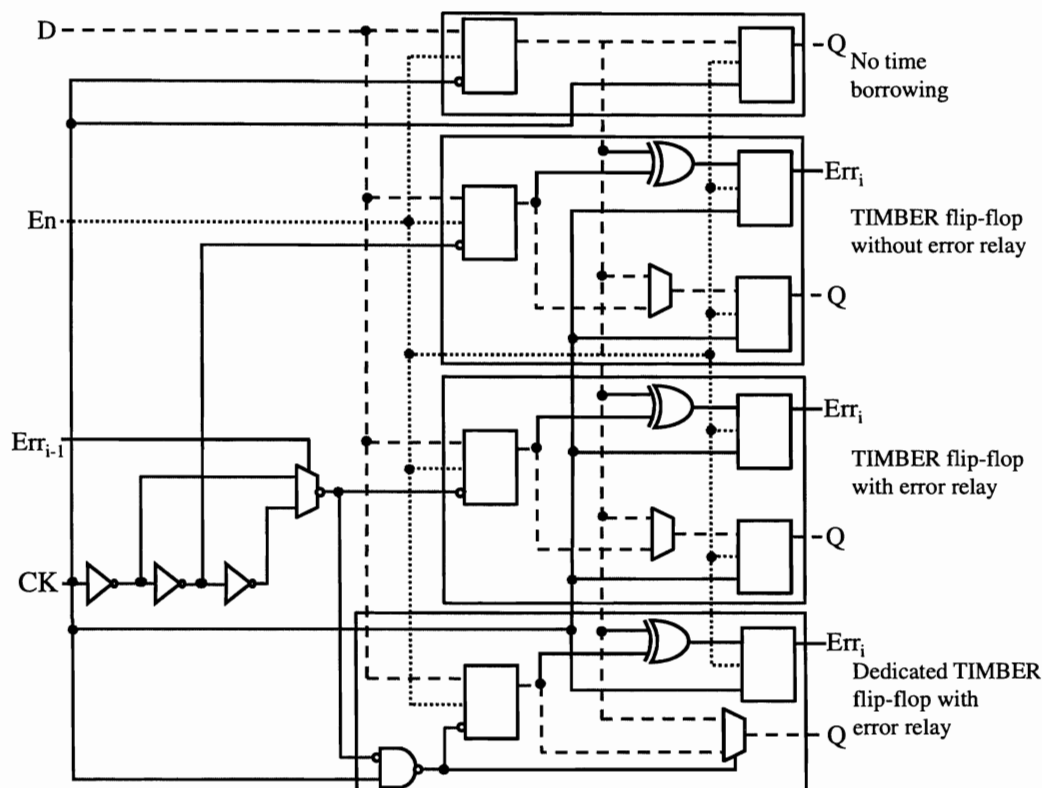


Figure 5.12 : FPGA implementation of TIMBER flip-flop.

latch from the circuit design of the TIMBER flip-flop.

The integrated TIMBER flip-flop has seven outputs — four data outputs and three error outputs. The four data outputs correspond to the four types of flip-flops described above. The three error outputs indicate a timing error during normal operation by comparing the outputs of the master latches for the flip-flops with time-borrowing to the master latch for the flip-flop without time-borrowing. The data output of the TIMBER flip-flop with error relay is used as the inputs to the next pipeline stage and its error output is used to relay errors to the next pipeline stage.

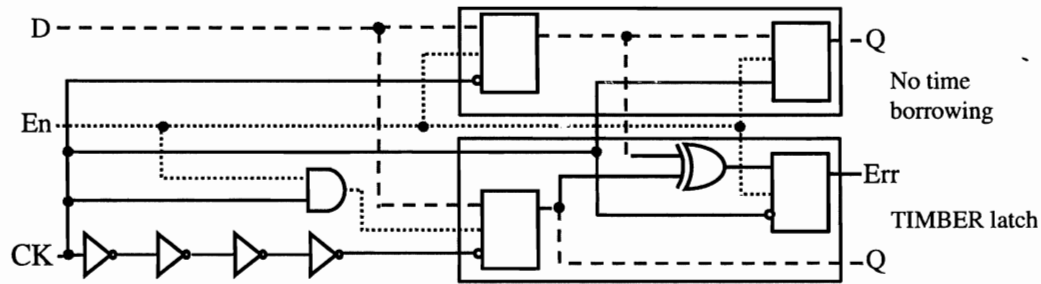


Figure 5.13 : FPGA implementation of TIMBER latch.

**Integrated TIMBER latch:** The integrated TIMBER latch consists of a flip-flop without time-borrowing and a TIMBER latch. The TIMBER latch provides continuous time-borrowing in each pipeline stage and a maximum time interval of four LUT delays can be borrowed in each pipeline stage. The integrated TIMBER latch has two outputs — two data outputs and one error output. The two data outputs corresponds to the flip-flop without time-borrowing and the TIMBER latch. The error output indicates a timing error during normal operation by comparing the output of the TIMBER latch to the output of the master latch for the flip-flop without time-borrowing.

Note that the error outputs generated by the integrated TIMBER flip-flop and TIMBER latch are used to detect timing errors during runtime. The error outputs from various flip-flops are consolidated at a central error control unit that scales frequency dynamically to mitigate multi-stage timing errors. Since frequency scaling using DCM requires re-compiling a design, our hardware setup does not include a central error control unit to scale frequency during runtime. When re-compiling for

different frequencies we did not enforce timing constraints to ensure that design timing remains unchanged at different frequency of operations.

The error outputs in the integrated flip-flop/latches will not be the correct timing error rate at higher frequencies when there are timing errors at the TIMBER flip-flop/latch. Although such high frequencies of operation will never be used during normal operation, to report accurate timing error rates in our hardware setup, we compare the data outputs of TIMBER flip-flops and TIMBER latch to a value obtained using a multi-cycle computation using the red XOR gates shown in Fig. 5.11. Thus, the purpose of incorporating error outputs into the integrated sequential elements is to accurately emulate a TIMBER flip-flop/latch that will be used in a real design.

**Timing description:** Our test structure uses 16 clock cycles to process each input pattern that is applied to the two-stage pipeline. Denote the 16 clock cycles as  $0 \cdots 15$  (see Fig. 5.14). The 32-bit LFSR generates two 16-bit operands as inputs to the multiplier on the  $5^{th}$  clock cycle. The outputs of the multiplier are latched for a single cycle computation on the  $6^{th}$  clock cycle. Similarly, the outputs of the second pipeline stage (32-bit XOR tree) are latched for a single cycle computation on the  $7^{th}$  clock cycle. The clock enable signals EN1 and EN2 are used as the enable inputs to the integrated sequential elements to ensure single cycle computation for the two pipeline stages. Multi-cycle computation is compared to the single cycle computation value on the  $10^{th}$  clock cycle for the first pipeline stage and on the  $15^{th}$  clock cycle for the

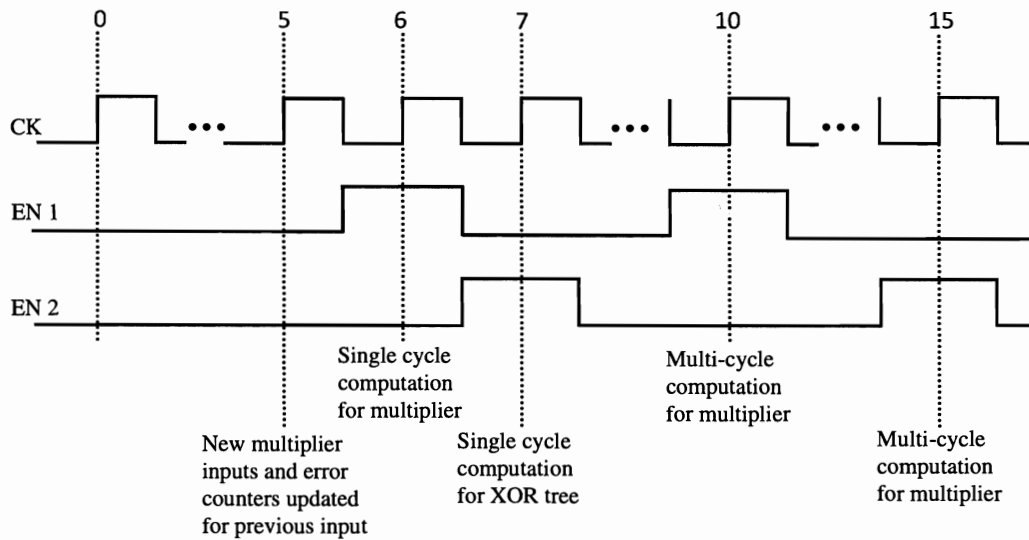


Figure 5.14 : Timing diagram for timing error rate measurement.

second pipeline stage using the red XOR gates as shown in Fig. 5.11. Fig. 5.11 shows only one XOR gate per integrated sequential element, but our implementation has four XOR gates for an integrated TIMBER flip-flop to compare multi-cycle computation to the data outputs of the four types of flip-flops contained in the integrated flip-flop design. Similarly, our implementation has two XOR gates for an integrated TIMBER latch. Note that the integrated flip-flop data outputs are updated to the correct value using a multi-cycle computation on the 10<sup>th</sup> clock cycle for the first pipeline stage and on the 15<sup>th</sup> clock cycle for the second pipeline stage.

A comparison of the timing error rates at different frequencies for a TIMBER flip-flop is shown in Fig. 5.15. We observe that the timing error rate saturates and remains constant after a certain frequency. Hence, we measured timing error rates for each flip-flop/latch design upto the frequency at which the timing error rate saturated.

The timing errors in Fig. 5.15 are reported as the total number of timing errors at the three outputs under observation normalized by the total number of timing errors occurring at the highest frequency. Timing errors for a conventional flip-flop without time borrowing are first observed at 60MHz. Some of these timing errors that occur at the output of the first pipeline stage are single-stage timing errors. However, the timing errors that occur at the output of the second pipeline stage contain both single-stage and two-stage timing errors. A two-stage timing error occurs when a timing error at the first pipeline stage causes the correct input to the second pipeline stage to be delayed, resulting in a timing error at the output of the second pipeline stage. As the frequency is increased, the first timing error for a TIMBER flip-flop without error relay occurs at the output of the second pipeline stage at 64.3MHz. However, these timing errors do not occur at the output of a TIMBER flip-flop with error relay, thus indicating that these errors are not single-stage timing errors. These errors are multi-stage timing errors that are masked by a TIMBER flip-flop with error relay logic because a timing error at the output of the first pipeline stage is relayed to allow the second pipeline stage to borrow additional time for masking multi-stage timing errors. When the frequency is increased further to 68.2MHz, the first timing error is observed at the output of a TIMBER flip-flop with error relay and at the output of a dedicated TIMBER flip-flop. At this frequency, single-stage timing errors occur at the output of the first pipeline stage because the timing violations exceed the maximum time-borrowing allowed in the design.

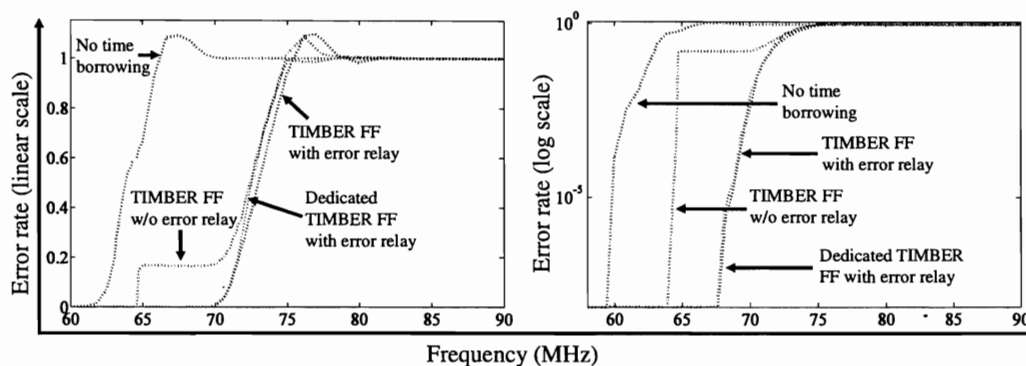


Figure 5.15 : Error rate vs frequency in linear scale and log scale for TIMBER flip-flop.

A comparison of timing error rates at different frequencies for a TIMBER latch is shown in Fig. 5.16. The first timing error for a conventional flip-flop without time-borrowing occurs at 60MHz and the first timing error in a TIMBER latch occurs at 76.2MHz. Note that the first timing error occurs at a higher frequency in a TIMBER latch than in a TIMBER flip-flop. This is because (i) unlike a TIMBER flip-flop, a TIMBER latch does continuous time-borrowing and hence small timing violations require borrowing small time intervals and (ii) the multiplexer in a TIMBER flip-flop for selecting between the master latches is implemented on an FPGA using an LUT. Thus, a TIMBER flip-flop has one LUT delay larger than the delay of a conventional flip-flop, that translates to a decrease in the time borrowing interval. In a custom design, this problem can be mitigated by using transmission gates or tri-state inverters to implement multiplexing.

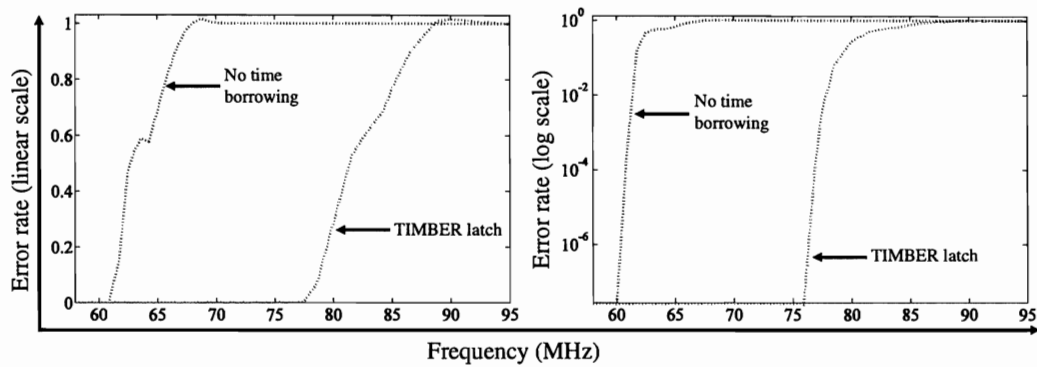


Figure 5.16 : Error rate vs frequency in linear scale and log scale for TIMBER latch.

## 5.5 TIMBER case study

We present results from a case-study when TIMBER is integrated into an industrial processor. Three processor performance points — low, medium, and high — each with four checking periods of 10%, 20%, 30%, and 40% of the clock period are considered. For a checking period equal to  $c\%$  of the clock period, all flip-flops terminating at the top  $c\%$  critical paths are replaced by a TIMBER sequential circuit element (TIMBER flip-flop or TIMBER latch). As described in Sec. 5.2, larger dynamic variability timing margins can be recovered for the same checking period by eliminating the TB interval, i.e., by flagging a single-stage timing error to the central error control unit. We present results for both cases in this section. When the TB interval is not considered, the checking period is divided into two ED intervals. Hence, for a checking period equal to  $c\%$  of the clock period, the timing margin recovered is equal to  $c/2\%$  of the clock period. When the TB interval is considered, the checking period is divided into one TB and two ED intervals. Thus, the timing margin recovered is



equal to  $c/3\%$  of the clock period. Results for both cases (without and with the TB interval) for the processor based on TIMBER flip-flop and TIMBER latch is reported in Fig. 5.10(i)-(ii) and Fig. 5.10(iii), respectively.

**TIMBER flip-flop:** The overhead for a design based on TIMBER flip-flop includes (i) overhead of a TIMBER flip-flop over a conventional flip-flop and (ii) overhead of the error relay logic. Fig. 5.10(i-a) shows the area overhead for the error relay logic used in TIMBER flip-flop architecture for the four checking periods at each performance point. Recall from Sec. 5.3 that the error relay logic can have a maximum latency of half a clock cycle. Fig. 5.10(i-b) presents the timing slack as the percentage of half the clock period for the error relay logic in the TIMBER flip-flop architecture. A large timing slack is available because error relay has to be performed only from a small number of TIMBER flip-flops that are the start and end-points of critical paths (refer Fig. 5.1). The total power consumption of a TIMBER flip-flop is about two times that of a conventional master-slave flip-flop. The switching activity in the error relay logic is small because under normal operation, the inputs to the error relay logic are all zeros and change only when a timing error occurs. Hence, the error relay logic mainly contributes to the static power overhead. Fig. 5.10(ii-a) and (ii-b) present the total power overhead for TIMBER flip-flop architecture over the base design without and with the TB interval, respectively.

**TIMBER latch:** The overhead for a design based on TIMBER latch can be attributed to the overhead of a TIMBER latch over a conventional master-slave flip-

flop. The total power consumption for a TIMBER latch is about 1.5 times that of a conventional master-slave flip-flop. Fig. 5.10(iii-a) and (iii-b) present the total power overhead for TIMBER latch architecture over the base design without and with the TB interval, respectively.

**TIMBER versus RAZOR:** RAZOR is a technique [19] that leverages delay variations due to dynamic variability and workload to reduce power consumption by dynamically scaling supply voltage and frequency during normal operation. Since delay variations due to dynamic variability and workload can be significant, RAZOR detects timing errors arising due to data signal delays of up to half a clock period. RAZOR uses double sampling of the data signal [18] to detect timing errors and roll-back or local instruction replay to restore the correct state in the system.

On the other hand, TIMBER is designed to improve performance by mitigating timing margins added to compensate for dynamic variability effects in complex, high-performance microprocessors. Since these microprocessors typically have deep pipelines and complex control logic, roll-backs and instruction replays require extensive hardware support and incur significant performance penalty. TIMBER is designed to avoid roll-backs or instruction replays by masking timing errors arising due to dynamic variability effects and ensuring that the system state is never corrupted by a timing error. Since delay variations due to dynamic variability much less significant than delay variations due to workload, TIMBER masks timing errors arising due to data signal delays up to 20% of the clock period.

## Chapter 6

### Performance optimization with approximate circuits

Chapter 4 and Chapter 5 described how approximate circuits for the input subspace most vulnerable to errors can be used to improve reliability of logic circuits. Chapter 4 demonstrated that approximate circuits for the timing-critical input subspace can be synthesized to have a smaller critical path delay than the original logic circuit. This motivated us to investigate the application of approximate logic circuits as decomposition for reducing the delay of a given logic circuit.

Decomposition of Boolean functions, the process of splitting a given Boolean function into smaller Boolean functions, is a well-researched area with initial work that can be traced back to the 1950s [67]. Although it is widely acknowledged by researchers that decomposition techniques can significantly reduce the delay, area, and power of digital designs [35? ], the decomposition techniques proposed in literature are computationally demanding. As digital designs have increased in size and complexity, to attain scalability to large designs, state-of-the-art logic synthesis tools have favored the use of local heuristics, instead of decomposition, to optimize the delay, area, or power of a designs. Although computationally efficient, local transformations often result in sub-optimal designs because they explore only a small synthesis space and

are restricted by the structure of the initial netlist.

Performance optimization is an important step used in state-of-the-art logic synthesis tools to increase frequency of operation by reducing the critical path delay of a design. Even without area and power constraints, state-of-the-art logic synthesis tools cannot guarantee optimality of delay during logic synthesis because the delay optimization algorithms are based on heuristics and local logic transformations. In this chapter, we demonstrate that logic decompositions based on approximate logic circuits can be used to reduce the critical path delay, on average, by 10% over state-of-the-art logic synthesis tools. Although area and power consumption of the logic circuit increases, logic decomposition using approximate circuits is useful in high-performance computing systems for pushing the performance envelope.

## 6.1 Related work and its limitations

The performance optimization techniques proposed in literature can be broadly divided into two classes: (i) structure-based and (ii) decomposition-based. The earliest techniques for timing-driven optimization were based on restructuring critical paths to reduce circuit delay [68; 69; 70; 71]. Most structure-based techniques have used the transformation of a ripple carry adder into a fast implementation like the CLA, carry select adder or carry bypass adder as motivation for their techniques. The technique proposed in [68], called tree height reduction, uses a CLA as motivation to reduce the delay of the circuit by rescheduling computation along critical paths. The

technique presented in [69], called the generalized select transform, uses a carry select adder as a motivating example and proposes a technique that identifies late arriving signals, performs computation using both 0 and 1 as the value for the signal, and then uses that signal to select the correct output through a multiplexer. In [70], the carry bypass adder is used as motivation to propose the generalized bypass transform that reduces the critical path delay by adding redundant bypass paths and turning the critical paths into false paths. The false paths can then be eliminated without increasing the delay of the circuit using a technique presented in [71].

Decomposition-based techniques fundamentally differ from structure-based techniques in that they do not directly restructure the circuit. Instead, the circuit structure is changed as a result of changing the functionality of the internal nodes, while maintaining functional equivalence at the primary outputs. Decomposition-based techniques are capable of exploring a much richer design synthesis space, at higher computational cost, as compared to structure-based techniques. A decomposition-based technique using partial collapsing and simplification of nodes to reduce the delay is proposed in [72]. The technique proposed in [73] uses permissible functions to resynthesize sets of nodes that lie on the critical path to reduce the delay. In [26], additional redundant circuitry is added to compute the output on input patterns that sensitize the critical paths. This approach includes features of structure-based techniques, but suffers the following drawbacks. Since redundant logic is added in the form of bypass paths to the original circuit, the technique leads to a circuit with

a high area and/or power footprint. The improvements in delay are limited because the additional redundant logic is restricted to only implications of the original function. The scalability of this approach is also limited due to a bottom-up synthesis approach for the additional redundant logic starting from an incompletely specified Boolean function with a large don't care space. Decomposition-based techniques using structural properties of BDDs have also been proposed [74; 75; 33; 76; 34] for timing optimization and to this day are an active area of research.

Although not directly related to this work, circuit approximation techniques for performance optimization based on speculative computation [77] have also been proposed in literature. Performance optimization is achieved by reducing the delay of the circuit implementation by relaxing the Boolean specification of outputs to allow the simplified circuit to occasionally compute an incorrect value. When an incorrect value is computed, the error is corrected by a roll-back or local instruction relay. Speculative techniques proposed in literature leverage designer knowledge for simplifying a logic circuit, and hence, have only been applied regular circuit structures such as adders [30; 31], rename and issue logic [30], and not to irregular multi-level logic circuits.

This chapter proposes a decomposition-based timing-driven optimization technique using lookahead logic circuits. Unlike prior techniques, where the synthesis of the decomposition functions is potentially expensive, our technique has the advantage that the decomposition functions are discovered in the synthesized form. It can ex-

plain conversion of a ripple carry adder into several fast implementations including the carry lookahead, carry select, and carry bypass adders. Like most other timing-driven optimization techniques, it also complements existing logic optimization algorithms. In Section 6.2, we develop the theory of lookahead logic circuits, and in Section 6.3, we describe the synthesis algorithm for lookahead logic circuits.

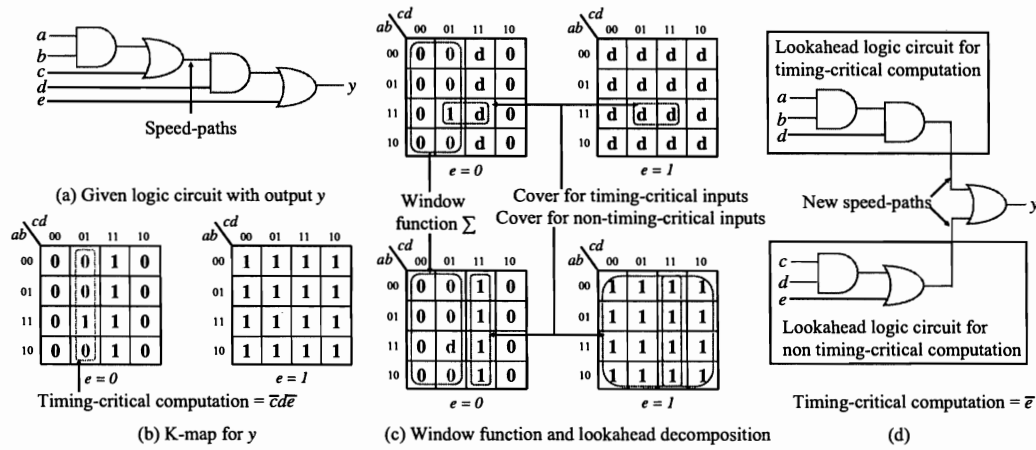


Figure 6.1 : Lookahead decomposition using timing-critical computation for general multi-level logic circuits.

## 6.2 Lookahead logic circuits

With the background on timing-driven optimization, we use binary addition to introduce the basic principles of prefix computation and then develop the theory of lookahead logic circuits. The most common approach to speed up carry computation in adders with large operand sizes is to exploit the observation that carry propagation in binary addition is a *prefix* problem [78].

### 6.2.1 Prefix problem

Given  $n$  values  $z_1, z_2, \dots, z_n$  and an associative binary operator  $\otimes$ , the prefix computation problem, or simply the prefix problem, is to compute the  $n$  values  $z_i \otimes z_{i-1} \otimes \dots \otimes z_1$ ,  $1 \leq i \leq n$ . In the context of binary addition of two  $n$ -bit numbers  $a$  and  $b$ , the carry for the  $i$ th bit can be expressed as a prefix problem as

$$c_i = G_{i:1} + P_{i:1}c_{\text{in}}, 1 \leq i \leq n \text{ where,}$$

$$(G_{i:j}, P_{i:j}) = (g_i, p_i) \otimes (g_{i-1}, p_{i-1}) \otimes \dots \otimes (g_j, p_j), i > j$$

where  $g_i = a_i b_i$  and  $p_i = a_i \oplus b_i$  represent the generate and propagate bits. The prefix element  $(g_i, p_i)$  and the prefix operator  $\otimes$  are given by

$$\text{Prefix element : } z_i = (g_i, p_i)$$

$$\text{Operator : } (u_1, v_1) \otimes (u_2, v_2) = (u_1 + v_1 u_2, v_1 v_2)$$

Thus, the carry out for the  $n$ -bit adder is given by

$$c_{\text{out}} = (G_{n:1}, P_{n:1}) = g_n + p_n g_{n-1} + \dots + p_n p_{n-1} \dots p_1 c_{\text{in}} \quad (6.1)$$

Since the prefixes  $g_i$  and  $p_i$  can be computed in parallel, the prefix problem reduces to efficient prefix computation and several tree structures, with size and depth trade-offs, have been proposed in literature to realize parallel-prefix adders [79].



We make the important observation that the parallel-prefix CLA can be thought of as an optimal timing-driven decomposition for carry computation. Consider the generalized Shannon decomposition of the carry for the  $i$ th bit given by

$$c_i = \Sigma_i(c_i)_0 + \overline{\Sigma}_i(c_i)_1 \quad (6.2)$$

where  $\Sigma_i$  is referred to as the window function, and  $(c_i)_0$  and  $(c_i)_1$  are the Boolean functions obtained by co-factoring  $c_i$  with respect to  $\Sigma_i$  and  $\overline{\Sigma}_i$ , respectively. The interesting connection between the CLA representation and the timing-driven decomposition lies in the expressions for  $\Sigma_i$ ,  $(c_i)_0$ , and  $(c_i)_1$ . Let us look at the timing-critical computation for the carry bit,  $c_i$ , of each stage of the  $n$ -bit adder. Note that  $c_i$  can be computed without the carry,  $c_{i-1}$ , of the previous stage when  $a_i = b_i = 0$  ( $c_i = 0$ ) and when  $a_i = b_i = 1$  ( $c_i = 1$ ). Thus, the case  $a_i = b_i$  is not a timing-critical computation at the  $i$ th bit-slice. However, when  $a_i \neq b_i$  ( $a_i \oplus b_i = 1$ ), the carry of the previous stage is necessary to compute  $c_i$ . Hence,  $a_i \neq b_i$  is a timing-critical computation at the  $i$ th bit-slice. By setting  $\Sigma_i = a_i \oplus b_i$ ,  $(c_i)_0 = c_{i-1}$ , and  $(c_i)_1$  to the value of  $c_i$  when  $\overline{\Sigma}_i = 1$ , i.e.,  $(c_i)_1 = a_i$  or  $(c_i)_1 = b_i$ , we obtain a timing-driven decomposition of  $c_i$  given by

$$c_i = (a_i \oplus b_i)a_i + (a_i \oplus b_i)c_{i-1}$$

The timing-driven decomposition for  $c_{\text{out}}$  of an  $n$ -bit adder can be obtained by ap-

plying this timing-driven decomposition to each bit-slice of an  $n$ -bit adder as follows

$$\begin{aligned}
 c_{\text{out}} &= (a_n \oplus \overline{b_n})a_n + \dots + (a_n \oplus b_n) \cdots (a_2 \oplus b_2)(a_1 \oplus \overline{b_1})a_1 + \\
 &\quad + (a_n \oplus b_n) \cdots (a_1 \oplus b_1)c_{\text{in}} \\
 &= a_n b_n + \dots + (a_n \oplus b_n) \cdots (a_2 \oplus b_2)a_1 b_1 + \\
 &\quad + (a_n \oplus b_n) \cdots (a_1 \oplus b_1)c_{\text{in}} \\
 &= g_n + p_n g_{n-1} + \dots + p_n p_{n-1} \cdots p_1 c_{\text{in}}, \tag{6.3}
 \end{aligned}$$

which is equivalent to the expression for  $c_{\text{out}}$  obtained using the prefix problem in equation 6.1.

The key contribution of the proposed timing-driven optimization technique is the use of information about timing-critical computation to identify window functions  $\Sigma_i$  that produce lookahead logic circuits  $(c_i)_0$  and  $(c_i)_1$  with fewer levels of logic. The regular modular structure of a binary adder makes it easy to identify a good timing-driven decomposition. However, in order to generalize and apply this technique to arbitrary multi-level control logic circuits, two main challenges must be addressed:

1. Extracting timing-critical computation: Multi-level logic circuits may have irregular structures that defy the easy modularity that makes it possible to write a CLA-like representation for the Boolean expression of the critical paths. The Boolean expression for the critical-path in multi-level logic may be significantly more complex, i.e., it cannot be expressed as a simple expression such as that

for the carry in adders.

2. Lookahead decomposition: Based on the information of timing-critical computation, the window function  $\Sigma_i$  and the cofactor functions  $(c_i)_0$  and  $(c_i)_1$  have to be constructed to realize a timing-driven decomposition with fewer levels of logic than the given circuit. The intuitive prefix structure of the adder made it easy to identify simple window and cofactor functions. However, a similar approach will not scale to arbitrary multi-level logic circuits.

We will now illustrate how timing-critical computation can be used to obtain lookahead decomposition for a general multi-level logic circuit. Consider the logic circuit with five inputs and an output  $y$  shown in Fig. 6.1(a). For a unit gate delay model, the two timing paths highlighted in red are the top 25% speed-paths of this circuit. Unlike an adder, where the timing-critical computation was identified by reasoning about its functionality, for a general multi-level logic circuit, we identify the timing-critical computation as the set of inputs that sensitize the speed-paths in the logic circuit. Fig. 6.1(b) shows the K-map for  $y$  and the inputs that constitute the timing-critical computation,  $\bar{c}d\bar{e}$ , are highlighted in red. The window function,  $\Sigma = \bar{c}\bar{e}$ , is constructed to contain timing-critical input space. Based on  $\Sigma$ , the lookahead decomposition  $y_0 = abd$  for the timing-critical computation and  $y_1 = cd + e$  for the non timing-critical computation are constructed. Note that the key challenge for a general multi-level logic circuit is to be able to use the information of timing-critical computation to construct  $\Sigma$ ,  $y_0$ , and  $y_1$  with fewer levels of logic. A timing-driven

decomposition for  $y$  is obtained by combining  $\Sigma$ ,  $y_0$ , and  $y_1$  using generalized Shannon decomposition.

$$y = \Sigma y_0 + \bar{\Sigma} y_1$$

$$y = \bar{c} \bar{e} (abd) + \bar{c} \bar{e} (cd + e)$$

Since  $y_0 \Rightarrow y$  and  $y_1 \Rightarrow y$ , the generalized Shannon decomposition can be simplified to  $y = y_0 + y_1$  (such simplifications are discussed in Section 6.3.3 and Table 6.1). Thus, the new timing-driven decomposition,  $y = abd + (cd + e)$ , is shown in Fig. 6.1(d). This new logic circuit has four speed-paths and the timing-critical computation has expanded to  $\bar{e}$ . Re-applying a timing-driven decomposition for the new logic circuit fails to further reduce the levels of logic. The following observations for this illustrative example are noteworthy.

- The identification of the timing-critical minterm,  $ab\bar{c}d\bar{e}$  in the on-set of  $y$ , plays an important role in the reduction of levels of logic for  $y$ . The  $\Sigma$ ,  $y_0$ , and  $y_1$  constructed from any other minterm in the on-set of  $y$  would not have reduced the levels of logic of the final decomposition.
- When the delay of the speed-paths in Fig. 6.1(a) are reduced, the delay of the short paths (primary input  $e$  to primary output  $y$ ) increase. This illustrates that our timing-driven optimization technique re-distributes computation from the critical paths to shorter paths. The increase in delay of short paths makes

satisfying hold time constraints easier, especially with overclocking techniques.

- To keep the illustration simple, we have restricted the lookahead decompositions,  $y_0$  and  $y_1$ , to be implicants of  $y$ . However, the algorithms described in this chapter can result in more general decompositions, e.g., the levels of logic for the lookahead decomposition  $y_1$  can be reduced by using  $y_1 = d + e$ . However,  $y_1 = d + e$  would break the implication relation  $y_1 \Rightarrow y$ . Hence, with  $y_1 = d + e$ , the generalized Shannon decomposition can longer be simplified, and the timing-driven decomposition for  $y$  will be  $y = \bar{c} \bar{e} (abd) + \overline{\bar{c} \bar{e}} (d + e)$ .

In the rest of this section, we describe algorithms for extracting the timing-critical computation in the form of the speed-path characteristic function and introduce the lookahead decomposition for arbitrary multi-level logic circuits.

### 6.2.2 Extracting timing-critical computation

For an output  $y$  of the circuit containing a critical path, the timing-critical computation can be extracted by identifying minterms in the input space of  $y$  that are responsible for exercising the speed-paths (critical or near-critical paths) terminating at  $y$ . These minterms are referred to as the timing-critical minterms or speed-path minterms in the input space of  $y$ . Recall from Chapter 4 that this set of minterms is referred to as the speed-path characteristic function (SPCF) for  $y$ . The SPCF is used to guide the synthesis of lookahead logic circuits.

Several algorithms have been proposed for the exact computation of the SPCF [26;

52]. These algorithms compute the exact set of minterms that sensitize paths with a delay greater than or equal to a desired value. These algorithms are path-based and require traversal of each critical path, which is memory and time intensive, especially when a complex and realistic gate delay model is used. To address issues of computational complexity in SPCF computation, algorithms that compute an approximation of the SPCF have also been proposed [53; 51]. These algorithms compute an over-approximation of the SPCF, i.e., minterms that do not sensitize critical paths may be included in the SPCF. The over-approximation algorithms are computationally more efficient than path-based algorithms because they compute the SPCF using a single forward traversal of the nodes on critical paths in the circuit. The sensitization minterms are computed for each node using the sensitization minterms for the nodes in the immediate fanin. Hence, the node-based techniques result in an over-approximation of the SPCF for circuits with reconvergent fanout. The details and illustrative examples for the over-approximation algorithm to compute the SPCF can be found in [51]. In [63], we extended the node-based algorithm from [51] to a path-based algorithm that computes the SPCF exactly. We showed that the computational complexity of path-based approaches to compute the SPCF can be reduced significantly by computing the SPCF based on the short path activation function, as opposed to the long path (i.e., critical path) activation function. The details of the algorithm, along with illustrative examples and comparisons to the other approaches to compute the SPCF, can be found in [63]. For the rest of this chapter, we assume

that for any logic circuit, the SPCF of all its outputs can be computed using either exact or approximation techniques proposed in literature.

### 6.2.3 Lookahead decomposition

Consider a Boolean function  $f(x_1, x_2, \dots, x_n)$  of  $n$  inputs  $x_1, x_2, \dots, x_n$ . Consider the decomposition for the Boolean function  $f$  given by the identity

$$f = \bar{\Sigma}_l f_l + \Sigma_l \bar{\Sigma}_{l-1} f_{l-1} + \dots + \Sigma_l \Sigma_{l-1} \dots \bar{\Sigma}_1 f_1 + \Sigma_l \Sigma_{l-1} \dots \Sigma_1 f_0 \quad (6.4)$$

where the window functions  $\Sigma_i$  and the cofactor functions  $f_i$  are all functions of  $x_1, x_2, \dots, x_n$ . By drawing an analogy to the CLA representation from equation 6.1, we can interpret the CLA representation from equation 6.4 as a *lookahead* decomposition for the Boolean function  $f$ . Here,  $\bar{\Sigma}_i f_i$  corresponds to the generate bit  $g_i$  and  $\Sigma_i$  corresponds to the propagate function  $p_i$ ,  $1 \leq i \leq l$ . However, lookahead decomposition of multi-level logic circuits is more complex than adders. Unlike an adder where both window and cofactor functions have a disjoint support set for  $1 \leq i \leq n$ , i.e.,  $\Sigma_i$  and  $\Sigma_j$  as well as  $f_i$  and  $f_j$  ( $i \neq j$ ) do not have common inputs in their support,  $\Sigma_i$  and  $f_i$  may not have disjoint support sets in multi-level logic circuits. Further, unlike an adder where the delay of each  $p_i$  and  $g_i$  term is equivalent to a single level of logic, the functions  $\Sigma_i$  and  $f_i$  may have different levels of logic and delays and hence combining them optimally presents challenges in multi-level logic circuits.

A simple functional approach for identifying window and cofactor functions will not scale for lookahead decomposition of multi-level logic circuits for the following reasons. First, there is no knowledge of the circuit implementation of  $\Sigma_i$  and  $f_i$ . Hence, a functional approach may result in a bad choice of  $\Sigma_i$  and/or  $f_i$  that may lead to a higher number of logic levels than the original circuit. Since the space of decompositions is vast, finding a good window and cofactor functions that can reduce the levels of logic for implementing  $f$  is challenging. Second, even with the knowledge of the functions  $\Sigma_i$  and  $f_i$  that can potentially reduce the levels of logic, directly synthesizing logic circuits for these Boolean functions is a challenge and does not scale as the complexity of these function increase.

In the next section, we will describe a synthesis algorithm that uses the information of timing-critical computation in the form of the SPCF to systematically simplify and reduce a given logic circuit to obtain lookahead logic circuits in the form of the window and cofactor functions,  $\Sigma_i$  and  $f_i$ , respectively.



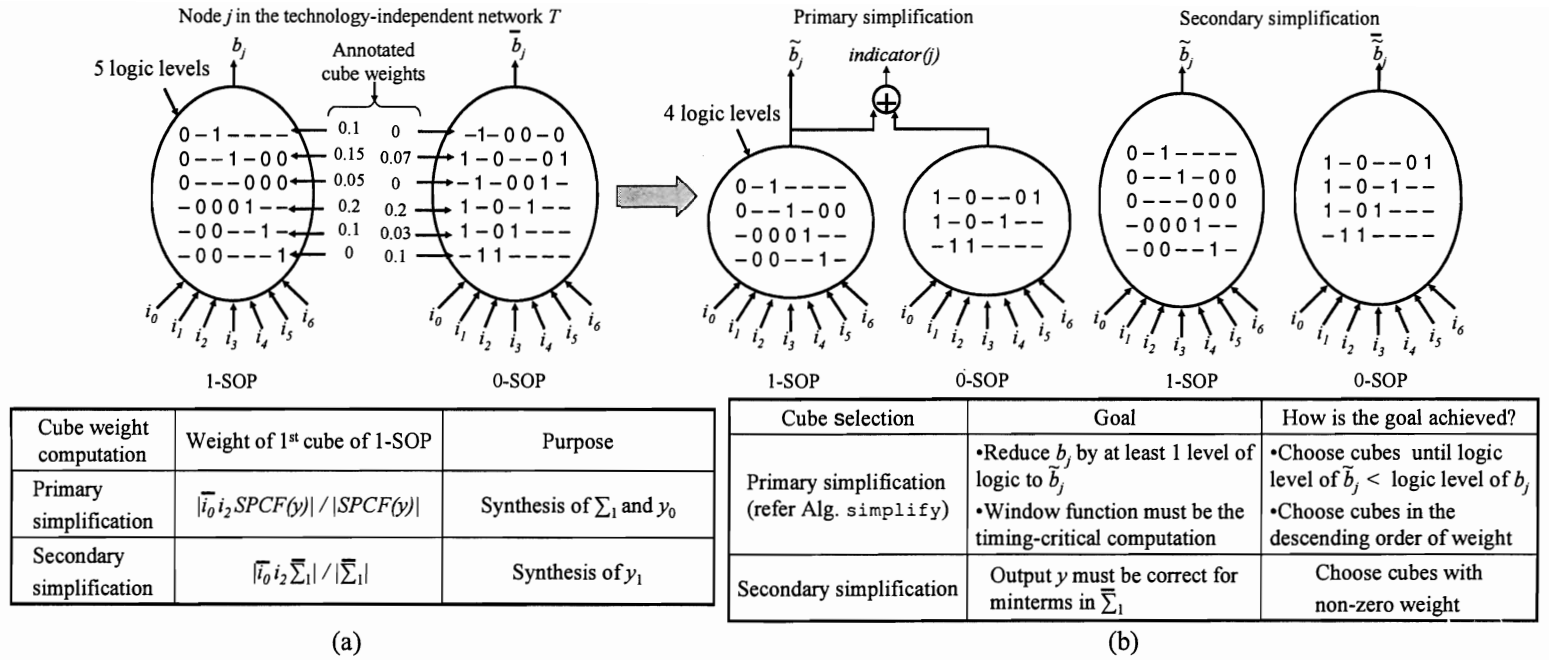


Figure 6.2 : Illustration of (a) cube weight computation and (b) cube selection used in primary and secondary simplification of the technology-independent network  $T$  during the synthesis of lookahead logic circuits.

### 6.3 Synthesis of lookahead logic circuits

Consider a decomposed circuit,  $\mathcal{C}$ , with  $n$  inputs,  $x_1, x_2, \dots, x_n$ , and  $m$  outputs. A decomposed logic circuit is a directed acyclic graph (DAG) with nodes representing AND gates. The edge connecting a node  $i$  to another node  $j$  can be of two types: (i) complemented, when there is an inverter between the output of node  $i$  and input of node  $j$  and (ii) uncomplemented, when there is no inverter. Thus, a decomposed circuit uses 2-input AND gates and NOT gates as building blocks, and is referred to as an and-invert-graph (AIG). An AIG can be converted into a technology-independent network, i.e., an intermediate DAG representation of a circuit in which the internal nodes are arbitrary Boolean functions using clustering algorithms (“renode” command in the tool ABC [37]).

Let  $l_{\mathcal{C}}$  denote the number of levels of logic in  $\mathcal{C}$ . Although our implementation considers all outputs simultaneously, for ease of notation and without loss of generality, we refer to a primary output  $y$  containing at least one critical path, i.e., at least one path with  $l_{\mathcal{C}}$  levels of logic for the rest of this discussion. The SPCF for  $y$  can be computed based on the algorithms described in Section 6.2.2. In this section, we will describe how the SPCF can be used to synthesize lookahead logic circuits starting from the given logic circuit  $\mathcal{C}$ . More specifically, we will describe the synthesis of the window function  $\Sigma_1$  and cofactor functions  $y_0$  and  $y_1$  for a single level

of timing-driven decomposition given by

$$y = \Sigma_1 y_0 + \bar{\Sigma}_1 y_1 \quad (6.5)$$

Further levels of timing-driven decompositions, performed either recursively or iteratively, will produce a set of window functions  $\Sigma_2, \Sigma_3, \dots, \Sigma_l$  and cofactor functions  $y_2, y_3, \dots, y_l$  analogous to Eqn. 6.4.

We propose a novel approach for the extraction of the functions  $\Sigma_1$ ,  $y_0$ , and  $y_1$  and the synthesis of their AIGs to have fewer logic levels than the original circuit. Our technique is based on two key ideas. First, we use transformations on the technology-independent network,  $\mathcal{T}$ , of the original decomposed circuit,  $\mathcal{C}$ , to synthesize the technology-independent networks for  $\Sigma_1$ ,  $y_0$ , and  $y_1$ . The transformations are made by simplifying the Boolean functions of the internal nodes in the technology-independent network to reduce the logic levels of the circuit. In this process, the functions  $\Sigma_1$ ,  $y_0$ , and  $y_1$  are derived dynamically during simplification. Second, we use the SPCF, extracted from the given decomposed circuit,  $\mathcal{C}$ , as a metric to guide the simplification. This ensures that the simplifications transform the functionality of the internal nodes significantly to reduce the levels of logic while using timing-critical computation to derive the window function  $\Sigma_1$  and the cofactor functions  $y_0$  and  $y_1$ .

The simplification of  $\mathcal{T}$  is performed in two stages. The first simplification, referred to as the primary simplification, is used to synthesize the technology-independent networks for  $\Sigma_1$  and  $y_0$  and the second simplification, referred to as the secondary

simplification, is used to synthesize the technology-independent network for  $y_1$ . Both primary and secondary simplifications involve simplifying the Boolean expressions of the internal nodes in  $\mathcal{T}$  by selecting a subset of cubes from the sum-of-product (SOP) expressions of their off-set (0-SOP) and on-set (1-SOP). As a result of the simplifications, the Boolean function for output  $y$  is transformed to  $y_0$  in the primary simplification and to  $y_1$  in the secondary simplification. In the primary simplification, additional logic for the technology-independent network of  $\Sigma_1$  is also added to  $\mathcal{T}$ . We will now describe the primary and secondary simplifications in detail using Fig. 6.2 as an illustrative example.

---

**Algorithm 2: simplify( $j$ )**


---

**input** :  $j$  is a node in  $\mathcal{T}$  with Boolean function  $b_j$  and logic level  $l_j$   
**output** :  $\tilde{b}_j$ , the simplified Boolean function for node  $j$   
 $S_0(S_1)$  is the minimum 0(1)-SOP of  $b_j$   
 $w(c)$  is the weight of cube  $c$ ,  $c \in S_0$  or  $c \in S_1$   
**if**  $w(c) = 0 \forall c \in S_0(S_1)$  **then**  
     $\tilde{b}_j = 0(1)$   
     $\mathcal{L}$  – Cubes of  $S_1$  ( $S_0$ ) in decreasing order of weight  
    **foreach**  $c \in \mathcal{L}$  **do**  
         $\tilde{b}_j(c) = 1(0)$   
        Compute **level** ( $j$ ) assuming  $\tilde{b}_j$  is the Boolean function of  $j$   
        **if** **level** ( $j$ )  $\geq l_j$  **then**  
             $\tilde{b}_j(c) = 0(1)$   
    indicator( $j$ ) =  $\tilde{b}_j$  ( $\overline{\tilde{b}_j}$ )  
**else**  
    Both 0-SOP and 1-SOP for  $j$  have non-zero weights  
    Initialize  $b_j^0 = 1$  and  $b_j^1 = 0$   
     $\mathcal{L}$  – Cubes of  $S_0$  and  $S_1$  in decreasing order of weight  
    **foreach**  $c \in \mathcal{L}$  **do**  
        **if**  $b_j(c) = 0$  **then**  
             $b_j^0(c) = 0$   
        **else**  
             $b_j^1(c) = 1$   
        Compute  $\tilde{b}_j$  assuming cubes added to  $b_j^0$  ( $b_j^1$ ) are in the off-set (on-set)  
        Compute **level** ( $j$ ) assuming  $\tilde{b}_j$  is the Boolean function of  $j$   
        **if** **level** ( $j$ )  $\geq l_j$  **then**  
            Remove  $c$  from  $b_j^0$  or  $b_j^1$  /\* Backtrack \*/  
    indicator( $j$ ) =  $\tilde{b}_j \oplus b_j$   
**mark** ( $j$ )

---



---

**Algorithm 3: reduce( $\mathcal{C}, \mathcal{T}, \text{SPCF}(l_{\mathcal{C}})$ )**


---

**input** : Decomposed circuit  $\mathcal{C}$  with  $l_{\mathcal{C}}$  levels of logic  
**input** :  $\text{SPCF}(l_{\mathcal{C}}) \forall$  output  $y \in \mathcal{C}$   
**input** : Technology-independent network  $\mathcal{T}$  for  $\mathcal{C}$  with  $l_{\mathcal{T}}$  levels of logic  
**output** : Modified  $\mathcal{T}$  with  $y_0$  and  $\Sigma_1 \forall$  output  $y \in \mathcal{C}$   
**foreach** output  $y$  of  $\mathcal{T}$  **do**  
    **if**  $\text{SPCF}(y) = 0$  **then**  
        **continue** /\* output does not contain critical path \*/  
    **repeat**  
         $\hat{j}$  = Unmarked node with highest logic level in **fanin** ( $y$ )  
         $b_j = \text{simplify}(j)$   
        Recompute logic level of nodes in  $\mathcal{T}$   
    **until** **level** ( $y$ )  $< l_{\mathcal{T}}$   
     $y_0 = y$  /\* output of the reduced network \*/  
     $\Sigma_1 = \bigwedge_{\text{marked nodes } j} (\text{indicator}(j))$   
    Unmark all nodes in  $\mathcal{T}$

---

Table 6.1 : Simplification rules for the generalized Shannon decomposition  $f = gf_g + \bar{g}f_{\bar{g}}$ .

Simplification rules for Shannon's decomposition $f = gf_g + \bar{g}f_{\bar{g}}$							
If	Then	If	Then	If	Then	If	Then
$\bar{g} \Rightarrow \bar{f}$	$f = gf_g$	$\bar{g} \Rightarrow f$	$f = \bar{g} + f_g$	$g \Rightarrow \bar{f}$	$f = \bar{g}f_{\bar{g}}$	$g \Rightarrow f$	$f = g + f_{\bar{g}}$
$\bar{f}_{\bar{g}} \Rightarrow \bar{f}$	$f = f_{\bar{g}}(\bar{g} + f_g)$	$\bar{f}_{\bar{g}} \Rightarrow f$	$f = f_g + \bar{g} + \bar{f}_{\bar{g}}$	$f_{\bar{g}} \Rightarrow \bar{f}$	$f = gf_g\bar{f}_{\bar{g}}$	$f_{\bar{g}} \Rightarrow f$	$f = gf_g + f_{\bar{g}}$
$\bar{f}_g \Rightarrow \bar{f}$	$f = f_g(g + f_{\bar{g}})$	$\bar{f}_g \Rightarrow f$	$f = \bar{f}_g + g + f_{\bar{g}}$	$f_g \Rightarrow \bar{f}$	$f = \bar{g}\bar{f}_gf_{\bar{g}}$	$f_g \Rightarrow f$	$f = f_g + \bar{g}f_{\bar{g}}$
$\bar{f}_g \Rightarrow \bar{f}$ $\bar{f}_{\bar{g}} \Rightarrow \bar{f}$	$f = f_gf_{\bar{g}}$	$\bar{f}_g \Rightarrow \bar{f}$ $\bar{f}_{\bar{g}} \Rightarrow f$	$f = f_g^\dagger$	$\bar{f}_g \Rightarrow \bar{f}$ $f_{\bar{g}} \Rightarrow \bar{f}$	$f = gf_g\bar{f}_{\bar{g}}$	$\bar{f}_g \Rightarrow \bar{f}$ $f_{\bar{g}} \Rightarrow f$	$f = gf_g + f_{\bar{g}}$ $f = f_g(g + f_{\bar{g}})$
$\bar{f}_g \Rightarrow f$ $\bar{f}_{\bar{g}} \Rightarrow \bar{f}$	$f = f_g^\dagger$	$\bar{f}_g \Rightarrow f$ $\bar{f}_{\bar{g}} \Rightarrow f$	$f = 1^\dagger$	$\bar{f}_g \Rightarrow f$ $f_{\bar{g}} \Rightarrow \bar{f}$	$f = g^\dagger$	$\bar{f}_g \Rightarrow f$ $f_{\bar{g}} \Rightarrow f$	$f = \bar{f}_g + g + f_{\bar{g}}$
$f_g \Rightarrow \bar{f}$ $\bar{f}_{\bar{g}} \Rightarrow \bar{f}$	$f = \bar{g}\bar{f}_gf_{\bar{g}}$	$f_g \Rightarrow \bar{f}$ $\bar{f}_{\bar{g}} \Rightarrow f$	$f = \bar{g}^\dagger$	$f_g \Rightarrow \bar{f}$ $f_{\bar{g}} \Rightarrow \bar{f}$	$f = 0^\dagger$	$f_g \Rightarrow \bar{f}$ $f_{\bar{g}} \Rightarrow f$	$f = f_{\bar{g}}^\dagger$
$f_g \Rightarrow f$ $\bar{f}_{\bar{g}} \Rightarrow \bar{f}$	$f = f_g + \bar{g}f_{\bar{g}}$ $f = f_{\bar{g}}(\bar{g} + f_g)$	$f_g \Rightarrow f$ $\bar{f}_{\bar{g}} \Rightarrow f$	$f = f_g + \bar{g} + \bar{f}_{\bar{g}}$	$f_g \Rightarrow f$ $f_{\bar{g}} \Rightarrow \bar{f}$	$f = f_g^\dagger$	$f_g \Rightarrow f$ $f_{\bar{g}} \Rightarrow f$	$f = f_g + f_{\bar{g}}$

### 6.3.1 Primary simplification of $\mathcal{T}$

Consider an internal node  $j$  in the fanin cone of output  $y$  in  $\mathcal{T}$  shown in Fig. 6.2. Let  $b_j$  denote the Boolean function of this node. Thus,  $b_j$  is a typical Boolean function with 10–15 inputs. Recall that the SPCF of  $y$ , denoted as  $SPCF(y)$ , contains the minterms that sensitize the critical paths that terminate at output  $y$ . The information of timing-critical computation represented in the SPCF is leveraged by assigning a weight to each prime implicant cube in the 0-SOP and 1-SOP of  $b_j$ .

**Cube weights:** The cube weight represents the fraction of minterms in the SPCF that will be covered if this cube is chosen during the simplification of this node. Hence, the weight of the cubes is the metric based on which the Boolean function of the internal nodes will be simplified. As an illustration, the weight of the first cube,  $\overline{i_0}i_2$ , of the 1-SOP of node  $j$  shown in Fig. 6.2 is given by  $|\overline{i_0}i_2SPCF(y)|/|SPCF(y)|$ . The cube weights can be computed using the global Boolean functions of each node and  $SPCF(y)$ . For computational efficiency, the cube weights for a node are computed only if the node is chosen for simplification by algorithm 3. We will now describe the simplification of a node based on the annotated cube weights.

**Simplify:** The main goal of the primary simplification of  $\mathcal{T}$  is to reduce the number of logic levels by simplifying the Boolean function of the internal nodes in  $\mathcal{T}$ . As we have seen in Section 6.2.1, having timing-critical minterms in the window functions can result in decompositions with fewer levels of logic. Thus, during primary simplification, the cubes that are important for timing-critical computation, i.e., the

cubes with larger weights, are chosen to be retained in the SOP expressions after simplification. The pseudo-code for the primary simplification algorithm, `simplify(j)`, is shown in algorithm 2. As an illustration, consider the internal node  $j$  shown in Fig. 6.2. Let  $\tilde{b}_j$  denote the Boolean function obtained after simplification of  $b_j$ .

During primary simplification,  $b_j$  is simplified to  $\tilde{b}_j$  and a signal `indicator(j)` is created to indicate when  $\tilde{b}_j$  matches  $b_j$ . The Boolean function,  $\tilde{b}_j$  and `indicator(j)`, are obtained by simplifying the off-set of  $b_j$  to  $b_j^0$  and the on-set of  $b_j$  to  $b_j^1$ . In other words, the functions  $b_j^0$  and  $b_j^1$  satisfy  $\overline{b_j^0} \Rightarrow \overline{b_j}$  and  $b_j^1 \Rightarrow b_j$ , respectively. The functions  $b_j^0$  and  $b_j^1$  are obtained by choosing cubes from the 0-SOP and 1-SOP of  $b_j$ , respectively (see Fig. 6.2). In order to capture as much timing-critical computation as possible, cubes from  $b_j$  are added to  $b_j^0$  and  $b_j^1$  in the decreasing order of their weights. After each cube is added to either  $b_j^0$  or  $b_j^1$ ,  $\tilde{b}_j$  and `indicator(j)` can be computed as  $\tilde{b}_j = b_j^0 = b_j^1$  and `indicator(j) =  $b_j^0 \oplus b_j^1$`  (see Fig. 6.2). Alternatively,  $\tilde{b}_j$  can be obtained by minimizing an incompletely specified Boolean function whose off-set and on-set is specified by the cubes in  $b_j^0$  and  $b_j^1$ , respectively. In this case, `indicator(j)` is given by the Boolean function  $b_j \oplus \tilde{b}_j$ . In our implementation, we explore both representations and choose the one with fewer levels of logic for  $\tilde{b}_j$ . The cubes are added to  $b_j^0$  and  $b_j^1$  until the levels of logic for  $\tilde{b}_j$  is less than the levels of logic for  $b_j$ .

The logic levels for the nodes in a technology-independent network is used during the simplification of the technology-independent network in the proposed algorithm and is also used to keep track of the progress in the reduction of the logic levels.



The logic level for a node  $j$ ,  $\text{level}(j)$ , is computed using the minimum sum-of-products (SOP) representation of the off-set and on-set for the Boolean function of node  $j$ . The minimum logic level is computed for the Huffman AND tree of each prime-implicant cube in the off-set and on-set. The minimum logic level for the Huffman OR tree is then computed using the minimum logic level of each cube. The smaller logic level value, between the off-set and the on-set, is defined as the logic level for node  $j$ . In addition, to computing the level of each node, the critical inputs can also be identified for each node. An input to a node is critical if the reduction of its level is a necessary condition for reducing the level of the node. The critical inputs to a node are also used in the function `reduce` to explore candidate nodes for the function `simplify`.

**Reduce:** The algorithm `reduce` determines which nodes in  $\mathcal{T}$  must be simplified. It also keeps track of the technology-independent node in  $\mathcal{T}$  that are simplified during the primary simplification by marking these nodes. In each iteration of the `while` loop in algorithm 3, an unmarked node with the highest level of logic is chosen for simplification. This is repeated until the at least one level of logic reduction is achieved over the original technology-independent network  $\mathcal{T}$  for the simplified technology-independent network of the cofactor function  $y_0$ . At the end of the primary simplification, some nodes in  $\mathcal{T}$  have been simplified to obtain a technology-independent network for  $y_0$  with less than  $l_C$  levels of logic. The window function,  $\Sigma$ , is the Boolean and of the indicator signals of all technology-independent nodes that were simplified

(see Algorithm 3.

### 6.3.2 Secondary simplification of $\mathcal{T}$

The primary simplification determines the window function  $\Sigma_1$  and the output  $y_0$ . In the secondary simplification,  $\mathcal{T}$  is reduced to generate the technology-independent network for  $y_1$ . Thus, in the secondary simplification, the complement of the window function,  $\overline{\Sigma}_1$ , is used to assign cube weights for the internal nodes. However, unlike the primary simplification, where the nodes had to be carefully chosen for simplification in order to obtain a good window function  $\Sigma_1$ , the only objective of the secondary simplification is to generate the technology-independent network for  $y_1$ . Hence, the objective is to reduce the levels of logic in  $\mathcal{T}$  as much as possible. This is done by replacing all cubes with zero weight by don't cares to simplify the Boolean function of every node.

After the technology-independent network for  $y_1$  is obtained using the secondary simplification, the original function  $y$  is reconstructed using the generalized Shannon decomposition and its implication based simplifications described in Section 6.3.3. A reduction in the levels of logic in the new decomposition concludes a single level of timing-driven decomposition. Otherwise, the algorithm 3 triggers backtracking by unrolling *all* simplifications because the primary simplification was not able to generate a window function to reduce the levels of logic for the cofactor function  $y_1$  obtained during the secondary simplification. Hence, by unrolling all simplifica-

tions, the original technology-independent network is restored and a different set of unmarked technology-independent nodes are explored for simplification.

### 6.3.3 Implication-based simplifications

In general, Eqn. 6.5 can be used to reconstruct  $y$  from  $\Sigma_1$ ,  $y_1$ , and  $y_0$ . However, the generalized Shannon decomposition expression can be simplified when  $\Sigma_1$ ,  $y_1$ , or  $y_0$  satisfies implication properties with  $y$ . We will now provide a few illustrations of these simplifications. In the following discussion, we have replaced  $\Sigma_1$ ,  $y_1$ , and  $y_0$  by  $g$ ,  $f_{\bar{g}}$ , and  $f_g$  for clarity of expressions. The three Boolean functions —  $g$ ,  $f_{\bar{g}}$ , and  $f_g$  — are involved in the generalized Shannon decomposition of  $f$ ,  $f = \bar{g}f_{\bar{g}} + gf_g$ . If only one of these three functions has an implication relation with  $f$ , then there are 12 possible implication relations with  $f$ . This is because four implication relations are possible between  $g$  and  $f$ , namely  $\bar{g} \Rightarrow \bar{f}$ ,  $\bar{g} \Rightarrow f$ ,  $g \Rightarrow \bar{f}$ , and  $g \Rightarrow f$ . Similarly, four implication relations are possible between  $f_{\bar{g}}$  and  $f$  and between  $f_g$  and  $f$ . The simplification of the Shannon decomposition based on these 12 implication relations with  $f$  are listed in the first three rows of Table 6.1.

If two of the three functions —  $g$ ,  $f_{\bar{g}}$ , and  $f_g$  — have an implication relation with  $f$ , then it may be possible to simplify the Shannon decomposition in two levels of simplification. However, when  $g$  has an implication relation with  $f$ , then after the first level of simplification using this implication relation, the Shannon decomposition is already simplified to an AND or OR decomposition, and thus, further simplification

is not possible. However, two levels of simplification is possible when  $f_g$  and  $f_{\bar{g}}$  have implication relations with  $f$ . There are 16 such implication relations since both  $f_g$  and  $f_{\bar{g}}$  have four possible implication relations with  $f$ . The simplification of Shannon decomposition based on these 16 implication relations are listed in the last 4 rows of Table 6.1.

All the simplifications shown in Table 6.1 can be derived using two results of Boolean algebra: (i)  $a + \bar{a}b = a + b$  and (ii) if  $f_1 \Rightarrow f_2$  then  $f_2 = f_1 + f_2$ . We have grouped the simplification rules into three categories based on similarity in their derivation — those marked with a †, those marked with a ‡, and the rest are unmarked. We will now present the derivation for one simplification of each category.

We start with an unmarked simplification rule: given the implication relations  $f_{\bar{g}} \Rightarrow f$  and  $f_g \Rightarrow f$ , the Shannon decomposition,  $f = \bar{g}f_{\bar{g}} + gf_g$  can be simplified to  $f = f_{\bar{g}} + f_g$ .

$$f = \bar{g}f_{\bar{g}} + gf_g \quad (6.6)$$

Since  $f_g \Rightarrow f$ ,  $f_g$  can be ORed to the right hand side of Eqn. 6.6. Thus,

$$\begin{aligned} f &= \bar{g}f_{\bar{g}} + gf_g + f_g \\ &= \bar{g}f_{\bar{g}} + f_g \end{aligned} \quad (6.7)$$

Similarly, since  $f_{\bar{g}} \Rightarrow f$ ,  $f_{\bar{g}}$  can be ORed to the right hand side Eqn. 6.7 to obtain

$$f = f_{\bar{g}} + f_g \quad (6.8)$$

Next, we describe the derivation of a simplification rule marked with a †. In these simplification rules, the Shannon decomposition of  $f$  simplifies to either  $g$ ,  $\bar{g}$ ,  $f_g$ , or  $f_{\bar{g}}$ . As an illustration, consider the implication relations  $\bar{f}_g \Rightarrow \bar{f}$  and  $\bar{f}_{\bar{g}} \Rightarrow f$ . Using  $\bar{f}_{\bar{g}} \Rightarrow f$ ,  $\bar{f}_{\bar{g}}$  can be ORed to the Shannon decomposition of  $f$  and simplified using  $a + \bar{a}b = a + b$  to obtain,

$$\begin{aligned} f &= \bar{g}f_{\bar{g}} + gf_g + \bar{f}_{\bar{g}} \\ &= \bar{g} + gf_g + \bar{f}_{\bar{g}} \\ &= \bar{f}_{\bar{g}} + \bar{g} + f_g \end{aligned} \quad (6.9)$$

Since the right hand side of Eqn. 6.9 is an OR of three functions, we can deduce that  $f_g \Rightarrow f$ . Combining  $f_g \Rightarrow f$  with the other implication relation,  $\bar{f}_g \Rightarrow \bar{f}$ , we obtain  $f = f_g$ .

Finally, we describe the derivation of a simplification rule marked with a ‡. In these simplification rules,  $f$  simplifies to a constant, i.e., the corresponding implication relations can never occur unless  $f$  is a constant. As an illustration, consider the implication relations  $\bar{f}_g \Rightarrow f$  and  $\bar{f}_{\bar{g}} \Rightarrow f$ . Using  $\bar{f}_g \Rightarrow f$ ,  $\bar{f}_g$  can be ORed to the

Shannon decomposition of  $f$  and simplified using  $a + \bar{a}b = a + b$  to obtain,

$$\begin{aligned}
 f &= \bar{g}f_{\bar{g}} + gf_g + \bar{f}_g \\
 &= \bar{g}f_{\bar{g}} + g + \bar{f}_g \\
 &= \bar{f}_g + g + f_{\bar{g}}
 \end{aligned} \tag{6.10}$$

Since the right hand side of Eqn. 6.10 is an OR of three functions, we can deduce that  $f_{\bar{g}} \Rightarrow f$ . However, the other implication relation requires  $\bar{f}_{\bar{g}} \Rightarrow f$ . The only way both these implications can be satisfied is when  $f = 1$ .

Our implementation of timing-driven optimization using lookahead logic circuits uses BDDs to check for implication relations of  $g$ ,  $f_g$ , and  $f_{\bar{g}}$  with  $f$ . Implication checks can also be done efficiently using SAT. For various benchmark circuits, we have observed that the implication-based rules are frequently used to reduce the number of levels of logic while reconstructing  $y$ . After reconstructing  $y$ , the technology-independent network for the reconstructed  $y$  is converted into a decomposed circuit by converting each node in the technology-independent network into an AIG. Area recovery is then performed using standard redundancy elimination algorithms.

#### 6.3.4 Fast adders

Historically, the adder has been an excellent example for evaluating various timing-driven optimization techniques primarily because of its regular prefix structure. Fast implementations of an  $n$ -bit adder include the (i) carry lookahead adder (CLA), (ii)

carry select or conditional carry adder, and (iii) carry bypass or carry skip adder. In Section 6.1, we have described how existing timing-driven optimization techniques have used one of these adders as a motivating example to develop timing-driving optimizations for general multi-level logic circuits. In contrast, our timing-driven optimization technique can be used to derive *all* these fast adders from a ripple carry adder. Let  $a$  and  $b$  be two 2-bit binary numbers and  $c_{\text{in}}$  be the carry-in bit. Let  $y$  denote the two bit sum and  $c_{\text{out}}$  denote the carry. Let  $g_i = a_i b_i$  denote the generate bit and  $p_i = a_i + b_i$  denote the propagate bit.

The simplest implementation of an adder is a ripple carry adder that can be realized by linearly cascading  $n$  full adders. Although the ripple carry-adder has a small area, the critical path delay of the ripple carry adder is  $O(n)$ . The carry-propagation logic is the most delay-intensive operation in a ripple carry adder. In a 2-bit ripple carry adder,  $c_{\text{out}} = g_2 + p_2(g_1 + p_1 c_{\text{in}})$  with 5 levels of logic. We will now explain how our timing-driven decomposition can transform a ripple carry adder into all these fast adders.

**CLA (4 levels, disjoint):** Based on the discussion in Section 6.2.1, two levels of timing-driven decomposition, i.e.,  $(\Sigma_2, y_2)$  and  $(\Sigma_1, y_1)$  can be used to convert a ripple

carry adder into a CLA. The window functions at the two levels are disjoint.

$$\begin{aligned}
 \Sigma_1 &= (a_1 \oplus b_1) \text{ and } \Sigma_2 = (a_2 \oplus b_2) \\
 y_0 &= c_{\text{in}}, y_1 = a_1, \text{ and } y_2 = a_2 \\
 c_{\text{out}} &= \bar{\Sigma}_2 y_2 + \Sigma_2 \bar{\Sigma}_1 y_1 + \Sigma_2 \Sigma_1 y_0
 \end{aligned} \tag{6.11}$$

**CLA (4 levels, overlapping):** The carry lookahead adder can also be obtained using a single-level of overlapping decomposition. The  $\Sigma_1$ ,  $y_0$ , and  $y_1$  obtained in this overlapping decomposition satisfies two implication relations with  $c_{\text{out}}$  —  $y_0 \Rightarrow c_{\text{out}}$  and  $y_1 \Rightarrow c_{\text{out}}$ . Thus, using the simplification rule described in Table 6.1, the Shannon decomposition of  $c_{\text{out}}$  can be simplified to  $c_{\text{out}} = y_1 + y_0$ .

$$\begin{aligned}
 \Sigma_1 &= (\bar{a}_1 + \bar{b}_1)c_{\text{in}} \\
 y_0 &= p_1 p_0 c_{\text{in}} \text{ and } y_1 = g_1 + p_1 g_0 \\
 c_{\text{out}} &= y_1 + y_0
 \end{aligned} \tag{6.12}$$

**Carry select and carry bypass adders (4 levels, overlapping):** For the carry select and carry bypass adders, a single-level of decomposition is sufficient to realize the final implementation. However, it is important to note that 2-bit carry select and carry bypass adders have 4 levels of logic if a multiplexer is considered as a single level of logic. Both decompositions are overlapping because  $y_1$  and  $y_0$  have common



inputs in their support. For the carry select adder, we have:

$$\begin{aligned}\Sigma_1 &= c_{\text{in}}, y_0 = g_2 + p_2 p_1, \text{ and } y_1 = g_2 + p_1 g_1 \\ c_{\text{out}} &= \overline{\Sigma}_1 y_1 + \Sigma_1 y_0\end{aligned}\tag{6.13}$$

For the carry bypass adder, we have:

$$\begin{aligned}\Sigma_1 &= p_2 p_1, y_0 = c_{\text{in}}, \text{ and } y_1 = g_2 + p_2 g_1 \\ c_{\text{out}} &= \overline{\Sigma}_1 y_1 + \Sigma_1 y_0\end{aligned}\tag{6.14}$$

**New decomposition (4 levels, overlapping):** The proposed technique also reveals another decomposition of the 2-bit adder with 4 logic levels. This decomposition also falls under the category of a single-level overlapping decomposition.

$$\begin{aligned}\Sigma_1 &= c_{\text{in}} + g_2 + p_2 g_1, y_0 = g_2 + p_2 p_1, \text{ and } y_1 = 0 \\ c_{\text{out}} &= \overline{\Sigma}_1 y_1 + \Sigma_1 y_0\end{aligned}\tag{6.15}$$

From these examples, it is clear that even a simple circuit like a 2-bit adder has four different decompositions with the optimal number of logic levels. This illustrates the expressive power of overlapping timing-driven decomposition techniques to extract equivalent descriptions with area-delay trade-offs.

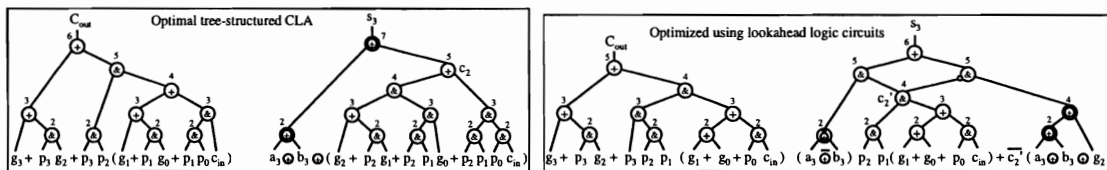


Figure 6.3 : Decomposition of  $c_{out}$  and  $s_3$  of a 4-bit adder in an optimal tree-structured CLA and using lookahead logic circuits.

## 6.4 Results

Our timing-driven optimization technique for synthesis of lookahead logic circuits is implemented within ABC [37]. All experiments were run on a 64-bit 2.4 GHz Opteron-based system with 6 GB memory. Our implementation uses binary decision diagrams (BDDs) for computing the SPCF and during the SOP-based cube selection during synthesis of lookahead logic circuits. Our implementation computes the exact SPCF using a path-based algorithm. For some circuits, the path-based algorithm for computing the SPCF may be computationally intensive. Since the SPCF is used only to guide the synthesis of lookahead logic circuits, for such circuits, over-approximate node-based techniques can also be used to speed-up computation of the SPCF. For some circuits, e.g., multipliers, BDD operations may be memory intensive. One approach to handle such circuits can be to optimize internal cones of logic containing critical paths instead of the entire circuit. However, for optimizing the entire circuit at once, further research is necessary to explore techniques based on SAT and logic simulation. Our approach has a runtime of 100 seconds on the largest circuit.

In Section 6.3.4, we illustrated the application of our timing-driven optimization

technique to transform the carry propagation logic of a 2-bit ripple carry adder into the carry propagation logic of carry lookahead adder and other high speed adders like carry select and carry bypass adders. We will present results for the application of our timing-driven optimization technique on complete  $n$ -bit adders,  $n \geq 4$ .

#### 6.4.1 Case study: $n$ -bit adder

In general, for an  $n$ -bit adder ( $n \geq 4$ ), identifying the adder implementation with the optimal number of logic levels is non-trivial. To illustrate this, we present the best results from SIS, ABC, an industry-standard synthesizer, and our technique to optimize an  $n$ -bit ( $n = 2, 4, 8, 16, 32$ ) ripple carry adder (details of the scripts used are given in the next section). We compare the results of synthesis to the theoretical optimum number of logic levels required to generate the carry in a tree-structured CLA for each value of  $n$  in Table 6.2. Note that in the optimum tree-structured CLA, the critical path terminates in the output computing the most significant bit (MSB) of the sum. Hence, the optimum number of logic levels for a 2-bit tree-structured CLA is 5, even though  $c_{out}$  has 4 logic levels. The number of logic levels obtained using existing techniques is higher than the theoretical optimum for the tree-structured CLA. In contrast, our technique matches the logic levels in an optimum tree-structured CLA for  $n = 2$  and discovers a circuit with one level of logic less than the optimum tree-structured CLA for  $n \geq 4$  as described below.

**4-bit adder:** We will now compare the decomposition of the 4-bit adder for the

Table 6.2 : Comparison of best AIG levels after timing optimization of an  $n$ -bit adder,  $n = 2, 4, 8, 16, 32$ .

$n$	Tree-structured CLA	SIS [80]	ABC [37]	Industry-standard synthesizer	Lookahead logic circuits
2	5	6	6	5	5
4	7	11	9	8	6
8	9	17	18	11	8
16	11	28	34	15	10
32	13	51	66	18	12

optimal tree-structured CLA with the decomposition obtained using lookahead logic circuits. Since our definition of logic levels is based on representing a circuit as an AIG, 2-input AND and OR gates have a single level of logic, but 2-input XOR and XNOR gates have two levels of logic. Thus, the computation of  $g_i = a_i b_i$  and  $p_i = a_i + b_i$  require one logic level. Figure 6.3 shows the decomposition for carry  $c_{\text{out}}$  and most significant sum bit  $s_3$  in a 4-bit adder for an optimal tree-structured CLA and for a circuit optimized using lookahead logic circuits. The decomposition of carry  $c_{\text{out}}$  in an optimal tree-structured CLA has 6 levels of logic and is reduced to 5 levels of logic when optimized using lookahead logic circuits. In the decomposition for carry  $c_{\text{out}}$  obtained using lookahead logic circuits,  $g_1$  is factored as  $p_1 g_1$  and this allows  $p_3 p_2 p_1$  to be factored out to obtain a more evenly balanced tree decomposition. This type of factoring is also used in Ling's high speed adder [81].

The decomposition for most significant sum bit  $s_3$  in the optimal tree-structured CLA is  $s_3 = a_3 \oplus b_3 \oplus c_2$ , where  $c_2$  is the carry out from third stage of the 4-bit adder. The decomposition of  $c_2$  in the optimal tree-structured CLA has 5 levels of logic, and thus,  $s_3$  has 7 levels of logic. The decomposition obtained using lookahead logic

circuits reduces one logic level over the optimal tree-structured CLA by balancing logic between  $a_3 \oplus b_3$  and  $c_2$ . This is done by separating  $g_2$  from  $c_2$  to obtain  $c'_2$ . In  $c'_2$ ,  $g_1$  is factored as  $p_1g_1$  and thus  $p_2p_1$  can be factored out to obtain a tree decomposition for  $c'_2$  with 4 levels of logic. Finally,  $s_3$  is decomposed using Shannon's decomposition as  $c'_2(a_3 \oplus b_3) + \overline{c'_2}(a_3 \oplus b_3 \oplus g_2)$  with 6 levels of logic. Thus, when  $c'_2 = 1$ ,  $c_2 = 1$  and  $s_3 = a_3 \oplus b_3$  and when  $c'_2 = 0$ ,  $c_2 = g_2$  and  $s_3 = (a_3 \oplus b_3 \oplus g_2)$ . For  $n > 4$ , the decompositions for  $c_{\text{out}}$  and  $s_3$  can be generalized to obtain  $n$ -bit adders with one level of logic less than the optimal tree-structured CLA.

Table 6.3 : Comparison of the proposed technique with the best algorithms in SIS, ABC, and the industry-standard synthesizer

Name	PI/POs	SIS [80]				ABC [37]				Industry-standard synthesizer				Lookahead logic circuits			
		Gates	Levels	Delay	Power	Gates	Levels	Delay	Power	Gates	Levels	Delay	Power	Gates	Levels	Delay	Power
x3*	135/99	822	9	81.5	2.6	617	12	97	2.3	693	8	85.4	2.3	669	8	80.5	2.27
alu2 <sup>+</sup>	10/6	482	22	187.8	2.1	349	32	237	1.9	367	18	184.7	2.1	349	12	114	1.6
alu4 <sup>+</sup>	14/8	841	24	214.4	4.05	637	33	251	3.9	655	22	226.8	4.1	926	19	178.2	4.8
apex4*	9/19	2194	12	145.7	5.3	2003	12	137.9	5.5	1718	12	138.3	5.2	1889	12	136.8	6.5
apex6*	135/99	664	11	104.3	2.2	602	14	110	2	681	10	104.8	2.1	670	8	80.6	2.35
dalv <sup>+</sup>	75/16	1604	20	227.4	2.8	1046	31	355.6	5	703	14	138.5	3.5	966	11	123.4	5.4
i10*	257/224	2454	29	436.2	12.6	1784	32	407.1	10.1	1935	26	284.4	13.9	1931	22	262	13.5
C432 <sup>+</sup>	36/7	273	22	260.4	2	136	23	248.7	1.2	197	19	205.4	2.4	250	15	174.4	3
C880 <sup>+</sup>	60/26	376	16	177.8	3	310	21	198.9	2.4	402	17	171	3.7	280	13	141.5	2.6
C2670 <sup>+</sup>	233/140	765	18	224.3	6.2	555	17	188.6	4.8	599	15	151.6	6.5	973	14	144.4	9.8
C5315 <sup>+</sup>	178/123	1784	21	245.4	13.3	1295	32	315.9	10.7	1464	26	261.8	14.1	1655	19	222	17.7
sparc_exu_ecl_flat*	572/634	2409	13	184.8	14.2	2108	13	161.4	14.5	2422	12	144.6	19.5	2191	11	150.1	20.1
lsu_stb_ctl_flat*	182/169	838	16	191.3	4.8	712	20	213.3	4.3	896	16	160.2	7.3	909	12	134.4	7.3
sparc_ifu_dcl_flat*	136/94	487	13	146.1	3.1	414	19	192.5	2.6	474	15	151.7	3.5	517	12	153.2	4
sparc_ifu_dec_flat <sup>†</sup>	131/146	881	14	153.3	4.4	797	14	158	4.4	923	13	186.4	6.2	828	13	151.8	6.6
lsu_excptl_flat*	251/179	670	12	130.9	4.9	567	13	137.6	4.3	685	12	133.7	5.8	743	12	127.7	6.6
sparc_tlu_intctl_flat*	82/80	227	8	96.5	1.2	174	11	115.4	1	304	7	77.7	2.6	266	6	76.9	2.5
sparc_ifu_fcl_flat*	465/522	2254	14	247.4	15.8	2043	17	256.1	13.8	2387	14	176.6	19.1	2459	11	184.4	22.4
tlu_hyperv_flat*	449/464	2397	17	198.9	14.5	2278	11	309.7	17	2573	11	128.6	20.3	2424	10	102.3	17
<b>Relative average</b>	–	1.17	1.095	1.22	0.88	<b>0.91</b>	1.33	1.3	<b>0.79</b>	1	1	1	1	1.01	<b>0.8</b>	<b>0.87</b>	1.086

<sup>+</sup> Arithmetic circuits \* Control circuits

<sup>†</sup> SIS: delay, rugged, algebraic and speed\_up; ABC: resyn2rs;

Industry-standard synthesizer: -map-effort high -area-effort high and set\_max\_delay 0

### 6.4.2 Benchmark circuits

We will now present results comparing timing-driven optimization using lookahead logic circuits to state-of-the-art academic tools SIS and ABC, and an industry-standard synthesizer. Fifteen circuits from the MCNC and ISCAS benchmark suites and the OpenSPARC T1 processor are used to compare our technique to the best results obtained using these tools. Each benchmark circuit is optimized with each tool — SIS, ABC, and industrial tool — to minimize the levels of logic. The optimized circuits are then mapped using the industrial tool to a gate library characterized with HSPICE for the 65nm CMOS predictive technology model. A load-dependent logical effort delay model is used to compute the delay of the mapped circuits. Our technique first maps the circuit to the gate library to extract timing-critical computation (or SPCF). Then, the levels of logic are reduced using lookahead logic circuits. Finally, the optimized circuit is re-mapped to the gate library. This process is repeated until no further improvements in delay are observed. For circuits optimized with each tool, an equivalence check is performed after optimization to ensure that the original and optimized circuits are equivalent. The first two columns in Table 7.1 give the circuit information. Subsequent columns report the number of gates in the AIG, logic levels in the AIG, technology-mapped delay, and the power consumption at 1GHz for the best results obtained with each optimization tool. Within SIS, the scripts `delay`, `rugged`, `algebraic`, and `speed_up` were used. For each benchmark circuit, the *best* results with the lowest technology-mapped delay are reported in the table. Within

---

ABC, script `resyn2rs` was used. Within the industry-standard synthesizer, each design was compiled with the options `-map-effort high` and `-area-effort high`. During synthesis, we set the design constraint `set_max_delay 0` to force the synthesizer to obtain minimum delay circuits. The last row in the table compares the tools, on average and normalized to the industry-standard tool. On average, our technique shows a 37%, 67%, and 20% reduction in the number of logic levels in the optimized circuit over SIS, ABC, and the industry-standard synthesizer, respectively. Note that, on average, the size of the decomposed circuit obtained using our technique and the industry-standard tool are comparable. When mapped delays are evaluated, our technique achieves an average reduction of 40%, 49% and 13% over the best results of SIS, ABC, and the industry-standard synthesizer, respectively. Thus, our technique provides a 13% improvement in performance for a 8.6% increase in the total power consumption over the industry-standard synthesizer. Four benchmark circuits — `alu2`, `i10`, `C880`, and `tlu_hyperv` — show a reduction in both delay and power. Since the reduction in the gate count for the benchmark circuits `alu2`, `i10`, and `tlu_hyperv`, is not significant and since we do not use power optimization steps during technology mapping, the power reduction in these circuits is possibly due to a smaller switching activity. For benchmark circuit `C880`, our technique significantly reduces both delay, gate count, and power. This is because our technique identifies a better decomposition for the benchmark circuit `C880`.



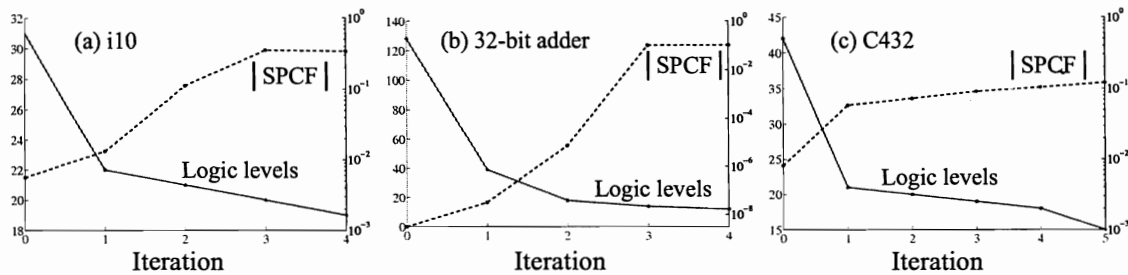


Figure 6.4 : In each figure, the left y-axis indicates the levels of logic and the right y-axis indicates the size of the SPCF as a fraction of the input space. Each circuit exhibits an increasing trend in the size of the SPCF when levels of logic are reduced using lookahead logic circuits.

### 6.4.3 Trend in SPCF

An interesting feature of timing-driven optimization based on lookahead logic circuits is that starting from a decomposition, the levels of logic are reduced progressively in several iterations. In each iteration, a new decomposition is obtained by separating the timing-critical input space from the non-timing-critical input space using the SPCF. We have observed that the size of the SPCF also increases progressively as the levels of logic in the decomposition are reduced. Figure 6.4 illustrates the increasing trend in the size of the SPCF as the delay of the decomposed circuit is optimized for single output cones from the benchmark circuits C432 and i10 with 194 and 853 gates, respectively, and the most significant bit of the sum output of a 32-bit adder. The x-axis indicates the iteration number for optimization, i.e., iteration 0 represents the initial decomposed circuit and the last iteration represents the final delay-optimized circuit. The y-axis on the left indicates the levels of logic of the circuit in each iteration and the y-axis on the right indicates the size of the SPCF as the fraction of

the input space for the top 10% critical paths. For each circuit, the size of the SPCF increases exponentially as the circuit is optimized and saturates at a value between 0.1–0.3 in the delay-optimized circuit. Hence, we believe that the size of the SPCF can be used as a metric to indicate how well a circuit has been optimized for delay.

This chapter described a timing-driven optimization technique based on lookahead logic circuits. Lookahead logic circuits are synthesized by simplifying the technology-independent network of the original circuit using critical path sensitization information. The original logic circuit is then reconstructed from the lookahead logic circuits using generalized Shannon decomposition and implication-based simplifications. Thus, new timing-driven decompositions of a circuit are explored by separating timing-critical computation from the non-timing-critical computation. The use of a technology-independent network for simplifications provides a computationally efficient means for searching a rich space of circuit decompositions to enhance the performance of the original circuit.

## Chapter 7

### Bi-decomposition of large Boolean functions

Chapter 6 demonstrated that logic decomposition based on approximate logic circuits can progressively push the performance of a given logic circuit by 10%, on average, over state-of-the-art logic optimization tools. However, we observed that for certain class of benchmark circuits, characterized by a large number of XOR gates, logic decomposition based on approximate circuits did not provide substantial reduction in critical path delay.

Since the timing-critical input sub-space was key to reducing the delay using approximate circuits, we decided to investigate the possibility of using the timing-critical input sub-space to identify and decompose XOR-friendly portions of the input space. This investigation revealed a simple, but powerful characteristic for determining the `and`, `or`, and `xor` bi-decomposability of Boolean functions. Based on this characteristic, this chapter describes a scalable algorithm for obtaining optimum variable partitions for bi-decomposition of Boolean functions by constructing an undirected graph called the blocking edge graph (BEG). Thus, although our initial goal was to develop a technique for reducing critical path delay for XOR-intensive circuits, our investigation resulted in a general algorithm for bi-decomposition of Boolean functions. To the best of our knowledge, this is the first algorithm that demonstrates

---

a systematic approach to derive disjoint and overlapping variable partitions for the bi-decomposition of large Boolean function.

## 7.1 Bi-decomposition overview and related work

Bi-decomposition, the simplest class of decomposition, recursively breaks down a Boolean function into two smaller Boolean functions. Bi-decomposition is an effective decomposition technique since it can be used to explore multi-level `and`, `or`, and `xor` decompositions. Bi-decomposition techniques rely on the ability to split the given logic function into two functions that depend on fewer input variables. Since the variable partition can significantly impact the quality of the decomposition, determining a good variable partition is not only the most important, but also the most computationally intensive step during bi-decomposition. Yang et al. and Wu et al. obtain a variable partition using the structural properties of a binary decision diagram (BDD), a canonical representation for the given logic function [33; 34]. However, their technique is memory intensive and sensitive to the variable order of the BDD used to extract the variable partition. Mishchenko et al. obtain a variable partition using heuristics that compromise the quality of the final decomposition [82]. Lin et al. use SAT solver coupled with interpolation techniques to reduce the runtimes for variable partitioning [83], but the variable partition obtained using this technique is still sub-optimal.

Given a logic function  $f$  with  $n$  inputs, we show that simple pairwise variable

co-factoring information can be used to derive a necessary and sufficient condition for a pair of variables to occur in the same partition of a bi-decomposition. Based on this condition, a BEG is constructed for `and`, `or`, and `xor` bi-decompositions of the logic function. We prove that a function is bi-decomposable *iff* the BEG for either the `and`, `or`, or `xor` bi-decomposition is not a complete graph. For bi-decomposable functions, we show that disjoint and overlapping variable partitions can be extracted by analyzing the vertex cuts of each BEG. Unlike existing approaches, optimal variable partitions with respect to two commonly used metrics (i) the total number of variables in the partitions and (ii) the size of the largest partition, can be obtained using BEGs. Furthermore, since a BEG has only one vertex per input variable of the function, variable partitioning based on BEGs is significantly faster than existing variable partitioning algorithm and is scalable to functions with several hundred inputs. Results indicate that on average, BEG-based bi-decomposition reduces the number of logic levels (mapped delay) of 16 benchmark circuits by 60%, 34%, 45%, and 30% (20%, 19%, 16% and 20%) over the best results of state-of-the-art tools FBDD, SIS, ABC, and an industry-standard synthesizer, respectively.

This chapter is organized as follows. Section 7.2 describes variable partitioning based on BEGs. Section 7.3 describes function bi-decomposition based on the variable partition identified using BEGs. Section 7.4 presents results.

## 7.2 Variable partition using blocking edge graphs (BEGs)

A function  $f$  of  $n$  variables is called *bi-decomposable* if it can be decomposed into two logic functions, each of which depends on less than  $n$  variables. The two smaller decomposed functions are combined using a two-input logic function. All two-input functions can be reduced to **and**, **or**, and **xor** operations upto complementation of inputs/output. Since any circuit with two-input gates can be reduced to a circuit with **and**, **or**, and **xor** gates by bubbling inverters down to the primary inputs, bi-decomposition techniques consider only **and**, **or**, and **xor** bi-decompositions of a logic function. Since **and** and **or** are dual operations, we obtain an **and** bi-decomposition for  $f$  from an **or** bi-decomposition of  $f$  by swapping the off-set and the on-set of  $f$  in this chapter.

Bi-decomposition techniques obtain smaller decomposed functions by first obtaining a variable partition of the variable set,  $V$ , of the given function  $f$ . A variable partition consists of two variable sets  $V_1$  and  $V_2$ , such that  $|V_1| < |V|$  and  $|V_2| < |V|$ . A variable partition is disjoint if  $V_1 \cap V_2 = \phi$ , otherwise the variable partition is overlapping. Variable partitions depend on the type of bi-decomposition — **and**, **or**, or **xor** — that we seek for the given logic function  $f$ . Hence, the most important and computationally intensive step during bi-decomposition involves determining the kind of decomposition and the variable partition for the given logic function.

Our technique uses undirected graphs called *blocking edge graphs (BEGs)* to extract variable partitions for **and**, **or**, and **xor** bi-decompositions of a logic function.

In this section, we first describe a necessary and sufficient condition, referred to as the *blocking condition*, for a pair of variables to be in the same variable partition of an and, or, or xor bi-decomposition of  $f$ . We then describe the steps for constructing separate BEGs for and, or, and xor bi-decompositions of a logic function based on the blocking condition. Finally, we show how variable partitions for and, or, and xor bi-decompositions can be extracted from the BEGs.

### 7.2.1 Blocking condition

For a pair of input variables,  $\{i, j\}$ , given a 0/1 assignment  $c$  of the variables in  $V \setminus \{i, j\}$ , the K-map of  $f$  can be restricted to a  $2 \times 2$  square covering the four cells  $c \cdot \bar{i} \bar{j}$ ,  $c \cdot i \bar{j}$ ,  $c \cdot \bar{i} j$ , and  $c \cdot i j$ . There are  $2^{n-2}$   $2 \times 2$  squares associated with the variable pair  $\{i, j\}$ , one for each 0/1 assignment of variables in  $V \setminus \{i, j\}$ . We classify the  $2 \times 2$  squares into 6 types based on the value of  $f$  in the four cells (see Fig. 7.2): (i) zero square with all four cells assigned 0, (ii) and square with three cells assigned 0, (iii) literal square with two adjacent cells assigned to 0, (iv) xor square with any two non-adjacent cells assigned to 0, (v) or square with three cells assigned to 1, and (vi) one square with all four cells assigned to 1. Note that although Fig. 7.2 shows only one  $2 \times 2$  square for each type, there are 4 different and squares, 4 different or squares, 4 different literal squares, and 2 different xor squares for a total of 16  $2 \times 2$  squares.

Given a logic function,  $f$ , of  $n$  variables and input variable set,  $V$  ( $|V| = n$ ), a variable partition,  $V_1$  and  $V_2$ , of  $f$  separates a pair of variables  $\{i, j\}$  if  $i \notin V_1$  and

$j \notin V_2$ . The variable pair  $\{i, j\}$  is *not separable* if there is no variable partition  $V_1$  and  $V_2$ , such that  $i \notin V_1$  and  $j \notin V_2$ . We will now describe a blocking condition for *and*, *or*, and *xor* bi-decomposition of  $f$ . The blocking condition for an *and* bi-decomposition is a necessary and sufficient condition for the a pair of variables  $\{i, j\}$  to be *not separable*. The blocking condition for *or* and *xor* bi-decompositions are defined in a similar manner. The blocking condition is derived based on the types of  $2 \times 2$  squares in  $f$  associated with a pair of variables  $\{i, j\}$ .

**Blocking condition:** The blocking condition for an *and* bi-decomposition of a completely specified logic function  $f$  states the following necessary and sufficient condition:  *$f$  has at least one  $2 \times 2$  or/xor square associated with a pair of variables  $\{i, j\}$  iff the variable pair  $\{i, j\}$  is not separable in an *and* bi-decomposition of  $f$ .* Similarly, *and/xor*  $2 \times 2$  squares block the separation of  $i$  and  $j$  in an *or* bi-decomposition and *and/or*  $2 \times 2$  squares block the separation of  $i$  and  $j$  in an *xor* bi-decomposition of  $f$ . The *zero*, *one*, and *literal* squares are non-blocking for *and*, *or*, and *xor* decompositions.

**Proof of blocking condition:** First, we will prove the forward implication of the blocking condition using contradiction. Suppose that  $f$  has at least one  $2 \times 2$  *or/xor* square associated with a pair of variables  $\{i, j\}$  and that there is a variable partition,  $V_1$  and  $V_2$ , for an *and* bi-decomposition of  $f$  that separates  $i$  and  $j$ . Without loss of generality, we assume that  $i \in V_1, j \notin V_1$  and  $j \in V_2, i \notin V_2$ . Suppose a  $2 \times 2$  *or/xor* square associated with  $\{i, j\}$  occurs for a 0/1 assignment,  $c$ , to the variables

---



in  $V \setminus \{i, j\}$ . Denote this  $2 \times 2$  or/xor square by  $f_c$ , a two variable function of  $\{i, j\}$ . Let  $g$  and  $h$  denote the decomposed functions for the variable partitions  $V_1$  and  $V_2$ , respectively. Let  $g_c$  ( $h_c$ ) be the single variable function of  $i$  ( $j$ ) obtained by assigning variables in  $V_1 \setminus \{i\}$  ( $V_2 \setminus \{j\}$ ) to their value in  $c$ . Since  $\{g, h\}$  is an and bi-decomposition of  $f$ ,  $g \cdot h = f$ . Hence,  $g_c \cdot h_c$  must also equal  $f_c$ . However, since  $f_c$  is an or/xor square, it cannot be covered by an and of a single variable function of  $i$  and a single variable function of  $j$ . Hence, this is a contradiction. Hence, there is no variable partition for an and bi-decomposition that separates  $i$  and  $j$ .

		$cd$			
		00	01	11	10
$ab$	00	0	0	0	0
	01	0	1	1	1
	11	1	1	0	0
	10	1	1	0	0

2 x 2  
squares  
for {b, d}

Figure 7.1 : Obtaining an and bi-decomposition from non-blocking squares.

Next, we will prove the reverse implication by proving the inverse of the blocking condition. In other words, we will show that if there is no  $2 \times 2$  or/xor square associated with a pair of variables  $\{i, j\}$  then there is a variable partition for an and bi-decomposition that separates  $i$  and  $j$ . If  $V$  is the variable set of  $f$ , we will show that  $V_1 = V \setminus \{i\}$  and  $V_2 = V \setminus \{j\}$  is a variable partition for an and bi-decomposition of  $f$ .

Consider the 4-input K-map shown in Figure 7.1. The 4  $2 \times 2$  squares associated with the variable pair  $\{b, d\}$  are marked with a solid line in the K-map. Note that we have chosen the K-map that contains all types except **or**/**xor**  $2 \times 2$  squares for  $\{b, d\}$ . For this K-map, we show the covering for each type of  $2 \times 2$  square to obtain an **and** bi-decomposition for the variable partition  $V_1 = \{a, b, c\}$  and  $V_2 = \{a, c, d\}$ . The cube cover with solid lines is a function of  $\{a, b, c\}$  and the cube cover with dotted lines is a function of  $\{a, c, d\}$ . Using the same type of cube cover, for any function  $f$  with variable set  $V$  that does not contain **or**/**xor**  $2 \times 2$  squares for  $\{i, j\}$ , an **and** bi-decomposition with variable partition  $V_1 = V \setminus \{i\}$  and  $V_2 = V \setminus \{j\}$  can be obtained. The blocking condition for **or** and **xor** bi-decompositions can be proved in a similar manner.

The blocking condition for **and**, **or**, and **xor** bi-decompositions are summarized in Figure 7.2. Based on the blocking condition, we will now describe a technique for extracting variable partitions by constructing a BEG for **and**, **or**, and **xor** bi-decompositions.

### 7.2.2 Constructing BEGs

A BEG has one vertex for each input variable of  $f$ . Hence, we will use  $V$  to denote both the input variable set of  $f$  and the vertex set of its BEG. In the BEG of an **and** decomposition, an edge is inserted between vertices  $i$  and  $j$  if the blocking condition for an **and** decomposition holds for the variable pair  $\{i, j\}$ . Similarly, an edge is

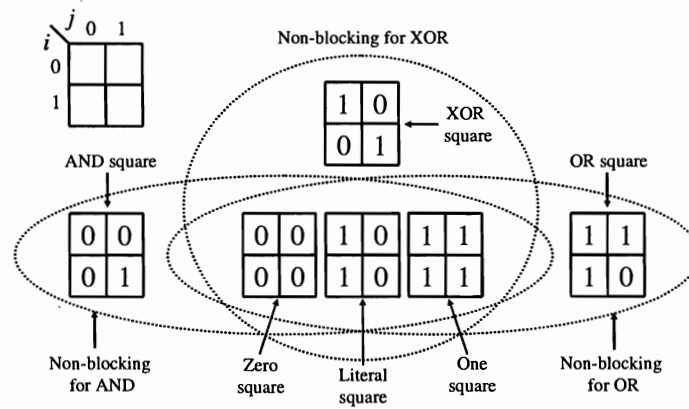


Figure 7.2 : Non-blocking squares for and, or, and xor bi-decompositions

inserted in the BEG of an or (xor) decomposition if the blocking condition for an or (xor) decomposition is satisfied for the variable pair  $\{i, j\}$ . Thus, an edge  $\{i, j\}$  in a BEG means that no variable partition can separate variable  $i$  and variable  $j$ .

For a logic function  $f$  with  $n$  variables, there are  $\binom{n}{2} 2^{n-2} 2 \times 2$  squares, i.e.,  $2^{n-2} 2 \times 2$  squares for each of the  $\binom{n}{2}$  variable pairs. We have developed an efficient algorithm for analyzing the types of these  $\binom{n}{2} 2^{n-2} 2 \times 2$  squares to enable fast construction of the BEGs for and, or, and xor bi-decompositions. Denote the off-set and on-set of a logic function  $f$  by  $f^0$  and  $f^1$ , respectively. Let the function  $x_{\{i,j\}}$  of  $n - 2$  variables in  $V \setminus \{i, j\}$  represent all  $2 \times 2$  xor squares associated with  $\{i, j\}$ , i.e., each minterm in  $x_{\{i,j\}}$  is a  $2 \times 2$  xor square associated with  $\{i, j\}$ . Similarly, let  $a_{\{i,j\}}$  and  $o_{\{i,j\}}$  represent the and and or squares associated with  $\{i, j\}$ , respectively. The

function  $x_{\{i,j\}}$  can be computed as follows:

$$x_{\{i,j\}} = \forall_i \forall_j (y^1 \cdot z^1) \quad (7.1)$$

$$\text{where } y^1 = f_i^0 \cdot f_i^1 + f_i^0 \cdot f_i^1 \text{ and } z^1 = f_j^0 \cdot f_j^1 + f_j^0 \cdot f_j^1$$

If  $x_{\{i,j\}}$  is not zero, then there are **xor** squares associated with  $\{i, j\}$  and hence, edge  $\{i, j\}$  is added in the BEG for the **and** and **or** bi-decompositions. Next, the functions  $a_{\{i,j\}}$  and  $o_{\{i,j\}}$  can be computed using functions  $y^1$  and  $z^1$  from Eqn. 7.1 as follows:

$$a_{\{i,j\}} = (\exists_i \exists_j (f^1 \cdot y^1 \cdot z^1)) \cdot (u \cdot v) \quad (7.2)$$

$$o_{\{i,j\}} = (\exists_i \exists_j (f^0 \cdot y^1 \cdot z^1)) \cdot (u \cdot v)$$

$$\text{where } y^0 = f_i^0 \cdot f_i^0 + f_i^1 \cdot f_i^1, z^0 = f_j^0 \cdot f_j^0 + f_j^1 \cdot f_j^1,$$

$$u = y_j^0 \cdot y_j^1 + y_j^1 \cdot y_j^0, \text{ and } v = z_i^0 \cdot z_i^1 + z_i^1 \cdot z_i^0$$

If  $a_{\{i,j\}}$  is not zero, then there are **and** squares associated with  $\{i, j\}$  and hence, edge  $\{i, j\}$  is added in the BEG for the **or** and **xor** bi-decompositions. Similarly, if  $o_{\{i,j\}}$  is not zero, then there are **or** squares associated with  $\{i, j\}$  and hence, edge  $\{i, j\}$  is added in the BEG for the **and** and **xor** bi-decompositions. The BEG for **and**, **or**, and **xor** bi-decompositions is constructed by computing  $x_{\{i,j\}}$ ,  $a_{\{i,j\}}$ , and  $o_{\{i,j\}}$  using Equations 7.1 and 7.2 for every pair of variables  $\{i, j\}$ .

### 7.2.3 Variable partition

In this section, we will show that the variable partitions of a logic function for the and, or, and xor bi-decompositions can be obtained by analyzing the connectivity of the BEGs. First, we provide a necessary and sufficient condition for the bi-decomposability of a function.

**Theorem 1:** *A logic function  $f$  is not bi-decomposable iff the BEG for the and, or, and xor bi-decompositions are complete graphs.*

**Proof:** If a logic function  $f$  is not bi-decomposable then there is no variable partition for and, or, or xor bi-decompositions that can separate any pair of variables. Hence, for every variable pair, the blocking condition is satisfied for and, or, and xor bi-decompositions. By construction, there is an edge between every pair of vertices in the BEG for and, or, and xor bi-decompositions, i.e, the BEG for and, or, and xor bi-decompositions are complete graphs. Since the blocking condition is necessary and sufficient, the converse can also be proved with the same argument.

Theorem 1 states that the bi-decomposability of a function  $f$  can be easily determined using BEGs. In the rest of this section, we describe how variable partitions can be obtained for bi-decomposable functions. We will describe our solution for decomposing functions that are not bi-decomposable in Section 7.3.2. However, we first describe two commonly used metrics used to measure the quality of a variable partition,  $V_1$  and  $V_2$ , of a logic function  $f$  with a variable set  $V$ ,  $|V| = n$ .

---

- Total variable count ( $\Sigma$ ): The total variable count,  $|V_1| + |V_2|$ , can range from  $n$  (for a disjoint decomposition) to  $2n - 2$  (for an overlapping decomposition with  $n - 2$  common variables and one unique variable per partition). Variable partitions with lower  $\Sigma$  are preferred since they typically result in decompositions with a small area and power footprint.
- Maximum partition size ( $\Delta$ ): The maximum partition size,  $\max(|V_1|, |V_2|)$ , can range from  $\lceil n/2 \rceil$  (for a balanced disjoint decomposition) to  $n - 1$  (since a bi-decomposition must produce functions that depend on less than  $n$  variables). Variable partitions with lower  $\Delta$  are preferred since they typically result in decompositions with low delay.

We will use  $\mu = [\Sigma, \Delta]$  to measure the quality of a variable partition. Measure  $\mu_1$  is less than measure  $\mu_2$  if either  $\mu_1(\Sigma) < \mu_2(\Sigma)$  and  $\mu_1(\Delta) \leq \mu_2(\Delta)$  or  $\mu_1(\Sigma) \leq \mu_2(\Sigma)$  and  $\mu_1(\Delta) < \mu_2(\Delta)$ . Thus,  $[2n - 2, n - 1]$  is the largest measure for a variable partition.

**Theorem 2:** *A bi-decomposable function  $f$  with variable set  $V$  has an and bi-decomposition with the overlapping variable partition  $V_1 = V \setminus \{i\}$  and  $V_2 = V \setminus \{j\}$  iff the edge  $\{i, j\}$  is not present in the BEG for the and bi-decomposition.*

**Proof: Proof:** Since  $V_1$  and  $V_2$  is a variable partition that separates  $i$  and  $j$ , the blocking condition for an and bi-decomposition is not satisfied for  $\{i, j\}$ . By construction, the BEG for an and bi-decomposition does not contain the edge  $\{i, j\}$ . The

converse of this theorem has been already proved in the converse of the blocking condition for an **and** bi-decomposition above. The proofs for **or** and **xor** bi-decompositions follow similarly.

Theorem 2 also holds for the **or** and **xor** bi-decompositions of  $f$ . Theorem 2 guarantees the existence of an overlapping variable partition for a bi-decomposable function and also shows how the overlapping variable partition can be obtained from the BEG of  $f$ . However, this variable partition may not be the best variable partition for  $f$  since it has the largest possible measure  $([2n - 2, n - 1])$ . Before we describe a technique for extracting better variable partitions from the BEGs of  $f$ , we review the definition of a vertex cut in graphs. A vertex cut of a connected graph is a set of vertices whose removal renders the graph disconnected. If  $C$  is a vertex cut of a graph with  $n$  vertices, then any super-set of  $C$  is also a vertex cut. The maximum size of a vertex cut is  $n - 2$ . Note that a complete graph with  $n$  vertices has no vertex cuts. A minimum vertex cut of a graph is the vertex cut with the smallest size. Note that a graph can have more than one minimum vertex cut. In this chapter, the vertex cut for a disconnected graph is assumed to be the empty set  $(\phi)$ .

**Theorem 3:** *If a bi-decomposable function  $f$  has an **and** bi-decomposition with the variable partition  $V_1$  and  $V_2$ , then  $V_1 \cap V_2$  is a vertex cut that disconnects the vertices in  $V_1 \setminus V_2$  from the vertices in  $V_2 \setminus V_1$  of the BEG for the **and** bi-decomposition.*

**Proof:** The variable partitions  $V_1$  and  $V_2$  separate every variable in  $V_1 \setminus V_2$  from every variable in  $V_2 \setminus V_1$ . Hence, the blocking condition is not satisfied for  $\{i, j\}$ ,  $i \in V_1 \setminus V_2$

and  $j \in V_2 \setminus V_1$ . Hence, the BEG for an **and** bi-decomposition does not contain the edge  $\{i, j\}$  and removing vertices in  $V_1 \cap V_2$  will disconnect the BEG for an **and** bi-decomposition. Hence,  $V_1 \cap V_2$  is a vertex cut whose removal disconnects vertices in  $V_1 \setminus V_2$  from the vertices in  $V_2 \setminus V_1$  in the BEG.

Theorem 3 also holds for the **or** and **xor** bi-decompositions of  $f$ . Using Theorem 3, the vertex cuts of the BEG can be used to obtain variable partitions for the **and**, **or**, and **xor** bi-decompositions of  $f$ . The minimum vertex cuts of the BEG can be used to obtain variable partitions with the smallest  $\Sigma$ . However, the variable partition obtained from minimum vertex cuts may have a large  $\Delta$  since the minimum vertex cut may disconnect the graph into components with unbalanced vertex set sizes. To reduce the value of  $\Delta$ , larger vertex cuts can be chosen (higher  $\Sigma$ ) that disconnect the graph into components with more balanced vertex set sizes.

Our solution to extract variable partitions for a function  $f$  starts with a list of minimum vertex cuts of the BEG for **and**, **or**, and **xor** bi-decompositions. The minimum vertex cut disconnects the BEG into smaller connected components. Larger vertex cuts are obtained by recursively augmenting the vertex cuts with the minimum vertex cut of the largest connected component. The vertex cut with the minimum value of  $\lambda\Sigma + \Delta$ , where  $\lambda$  is a parameter used that determines the relative importance of  $\Sigma$  and  $\Delta$ , is then chosen as the variable partition for  $f$ . The computational details of extracting the minimum vertex cut from an undirected graph are described in Section 7.4.



## 7.3 Function decomposition

In the previous section, we have described a technique based on BEGs for extracting variable partitions for `and`, `or`, and `xor` bi-decompositions of a bi-decomposable logic function  $f$ . The first part of this section describes the decomposition of a bi-decomposable function using a determined variable partition,  $V_1$  and  $V_2$ . The second part of this section describes the decomposition of functions that are not bi-decomposable.

### 7.3.1 Bi-decomposable functions

Denote the off-set and on-set of  $f$  by  $f^0$  and  $f^1$ , respectively, and the off-set and on-set of the decomposed functions for the variable partition  $V_1(V_2)$  by  $f_1^0(f_2^0)$  and  $f_1^1(f_2^1)$ , respectively. Given  $f^0$ ,  $f^1$ , and the variable partition,  $V_1$  and  $V_2$ , we will now describe how  $f_1^0$ ,  $f_1^1$ ,  $f_2^0$ , and  $f_2^1$  can be determined for `and`, `or`, and `xor` bi-decompositions.

**and/or bi-decompositions:** For an `or` bi-decomposition, the on-sets of the decomposed functions,  $f_1^1$  and  $f_2^1$ , are a subset of the on-set,  $f^1$ , of  $f$ . Hence,  $f_1^0$  and  $f_2^0$  can be obtained by expanding the off-set,  $f^0$ , of  $f$  using the existential operator over the variables in  $V \setminus V_1$  and  $V \setminus V_2$  as follows:

$$\begin{aligned} f_1^0 &= \exists_{V \setminus V_1} f^0 \\ f_2^0 &= \exists_{V \setminus V_2} f^0 \end{aligned} \tag{7.3}$$

Note that the off-sets of the decomposed functions may overlap with the on-set,  $f^1$ , of  $f$ . The on-set,  $f_1^0$  and  $f_2^0$ , for the decomposed functions are obtained by expanding

the portion of  $f^1$  that does not overlap with the off-set  $f_1^0$  or  $f_2^0$  of the decomposed functions, using the existential operator over the variables in  $V \setminus V_1$  and  $V \setminus V_2$ .

$$\begin{aligned} f_1^1 &= \exists_{V \setminus V_1}(f^1 \cdot \overline{f^0}) \\ f_2^1 &= \exists_{V \setminus V_2}(f^1 \cdot \overline{f^0}) \end{aligned} \tag{7.4}$$

An **and** bi-decomposition can be obtained in a similar manner by interchanging the off-set and the on-set of  $f$ .

**xor bi-decomposition:** An xor bi-decomposition requires more effort than an and/or bi-decomposition. To obtain an xor bi-decomposition of  $f$  for variable partitions  $V_1$  and  $V_2$ , we use an approach previously proposed in [84] to progressively grow the on-set and off-set of the decomposed functions by adding minterms to cover disjoint portions of the on-set of  $f$ . The pseudocode for the xor bi-decomposition is described in Algorithm 4.

**Infeasible variable partitions:** For certain functions, the variable partitions obtained from the vertex cuts of the BEG may be infeasible. A variable partition,  $V_1$  and  $V_2$ , for a function  $f$  is *infeasible* if  $f$  cannot be decomposed into smaller functions with variable sets  $V_1$  and  $V_2$ . The infeasibility of a variable partition of  $f$  can be detected when  $f$  cannot be obtained by composing the smaller sub-functions in the decomposition of  $f$ . For an infeasible variable partition for the or/and bi-decomposition, the smaller sub-functions will not cover the entire on-set (off-set) of  $f$ . For an xor bi-decomposition, if the on-set and off-set of the decomposed functions overlap at any

---

**Algorithm 4: xor bi-decomposition**


---

```

input      :  $f^0(f^1)$  is the off-set (on-set) of  $f$ 
input      :  $V, V_1, V_2$  are the variable set and the two variable partitions of  $f$ 
output     :  $f_1^0(f_1^1)$  is the off-set (on-set) for  $V_1$ 
output     :  $f_2^0(f_2^1)$  is the off-set (on-set) for  $V_2$ 

 $f_1^0 = 0, f_1^1 = 0, f_2^0 = 0, f_2^1 = 0$ 
 $g_1^0 = 0, g_1^1 = 0, g_2^0 = 0, g_2^1 = 0$ 
while ( $f \neq 0$ ) do
     $g_1^1 = \text{PickOneCube}(f^1)$ 
    while ( $g_1^0 + g_1^1 \neq 0$ ) do
         $g_2^0 = \exists_{V \setminus V_2}(f^1 \cdot g_1^1 + f^0 \cdot g_1^0)$ 
         $g_2^1 = \exists_{V \setminus V_2}(f^1 \cdot g_1^1 + f^0 \cdot g_1^1)$ 
        if ( $g_2^0 \cdot g_2^1 \neq 0$ ) then
             $\perp$  return  $f_1^0, f_1^1, f_2^0, f_2^1 = 0$  /* Variable partition infeasible */
         $f^0 = f^0 - (g_1^0 + g_1^1); f^1 = f^1 - (g_1^0 + g_1^1)$ 
         $f_1^0 = f_1^0 + g_1^0; f_1^1 = f_1^1 + g_1^1$ 
         $g_1^0 = \exists_{V \setminus V_1}(f^1 \cdot g_2^1 + f^0 \cdot g_2^0)$ 
         $g_1^1 = \exists_{V \setminus V_1}(f^1 \cdot g_2^1 + f^0 \cdot g_2^0)$ 
        if ( $g_1^0 \cdot g_1^1 \neq 0$ ) then
             $\perp$  return  $f_1^0, f_1^1, f_2^0, f_2^1 = 0$  /* Variable partition infeasible */
         $f^0 = f^0 - (g_2^0 + g_2^1); f^1 = f^1 - (g_2^0 + g_2^1)$ 
         $f_2^0 = f_2^0 + g_2^0; f_2^1 = f_2^1 + g_2^1$ 
    if ( $f^0 \neq 0$ ) then
         $f_1^0 = f_1^0 + \exists_{V \setminus V_1} f^0$ 
         $f_2^0 = f_2^0 + \exists_{V \setminus V_2} f^0$ 

```

---

point during Algorithm 4, then the variable partition is infeasible.

Variable partitions obtained using vertex cuts from a BEG are sometimes infeasible because Theorem 3 only mandates that a vertex cut of the BEG is a necessary, but not sufficient condition for a variable partition of  $f$ . For instance, in Figure 7.3(a), although the BEG for an or bi-decomposition for  $f$  indicates that a disjoint bi-decomposition exists,  $f$  only has an overlapping or bi-decomposition. Infeasibility of variable partition arises when a function has multiple variable partitions of the same cost. For the example shown in Figure 7.3,  $(\{a, b\}, \{b, c\})$ ,  $(\{a, b\}, \{a, c\})$ , and  $(\{a, c\}, \{b, c\})$  are feasible overlapping variable partitions for an or bi-decomposition of  $f$ . Since there is a variable partition that separates every variable pair, the blocking

condition is not satisfied for any variable pair. Hence, there are no edges in the BEG for an or bi-decomposition of  $f$ . However,  $(\{a\}, \{b, c\})$  is not a feasible variable partition.

In practice, for various benchmark circuits, we have observed that infeasible variable partitions are rare ( $< 5\%$  cases). Our technique handles an infeasible variable partition for a function by creating an overlapping variable partition,  $V \setminus \{i\}$  and  $V \setminus \{j\}$ , such that  $\{i, j\}$  is not an edge in the BEG. Note that such a  $\{i, j\}$  always exists since the BEG for  $f$  is not a complete graph and Theorem 2 guarantees the validity of the overlapping partition.

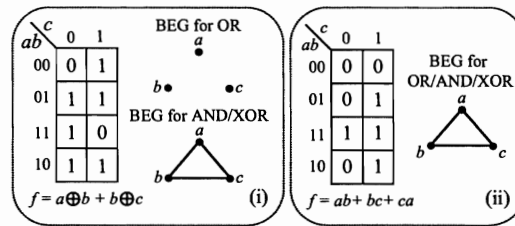


Figure 7.3 : (a) Incorrect disjoint decomposition indicated by BEG and (b) function with a complete BEG.

### 7.3.2 Non bi-decomposable functions

Recall that a function is not bi-decomposable if it cannot be decomposed into two functions that each depend on less than  $n$  variables. The BEG for the and, or, and xor bi-decompositions of these functions are complete graphs, and hence there are no vertex cuts for the BEGs. Figure 7.3(b) illustrates an example of a 3-input function that is not bi-decomposable. Our technique decomposes these functions using an or

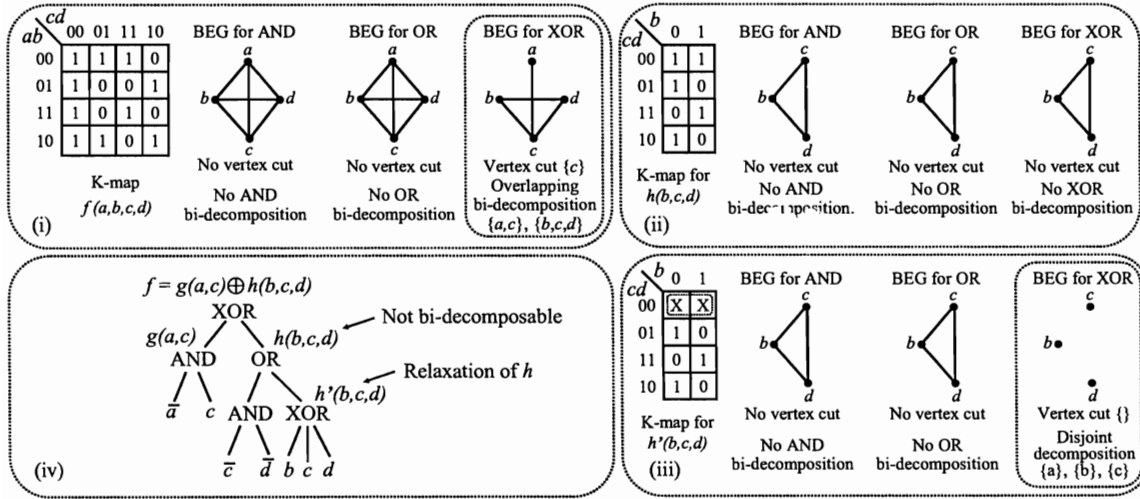


Figure 7.4 : Bi-decomposition using BEGs of (i)  $f(a,b,c,d)$ , (ii)  $h(b,c,d)$ , and (iii)  $h'(b,c,d)$ . (iv) summarizes the bi-decomposition.

decomposition. The first function of the or decomposition is obtained by relaxing  $f$  by introducing don't cares. Don't cares are introduced using a universal quantification of  $f$  with a variable  $i$  such that  $\forall_i f$  covers the minimum number of minterms in the on-set. Thus,  $\forall_i f$  is the don't care space for the relaxation of  $f$ . After decomposing the relaxation of  $f$ , the second function of the decomposition is setup to cover the portion of the on-set that was not covered by the first function.

## 7.4 Bi-decomposition results

We will start by illustrating our BEG-based bi-decomposition technique on the 4-input logic function shown in Figure 7.4(i). First, we construct the BEGs of  $f$  for and, or, and xor bi-decompositions. Since the BEG for and and or bi-decompositions are complete graphs, there are no and or or bi-decompositions for  $f$ . The BEG for

the **xor** bi-decomposition of  $f$  is not a complete graph and has the set  $c$  as a vertex cut of the BEG. Hence, there is an overlapping variable partition  $(\{a, c\}, \{b, c, d\})$  for an **xor** bi-decomposition of  $f$ , i.e.,  $f = g(a, c) \oplus h(b, c, d)$ . Using algorithm 4, it is determined that the variable partition is feasible and the decomposed functions  $g$  and  $h$  are also obtained. Since  $g$  is the simple two input function  $\bar{a}c$ , Figure 7.4 does not show the steps for the obtaining the decomposition of  $g$ .

The bi-decomposition of  $h$  is the next recursive step and is shown in Figure 7.4(ii). Since the BEG for the **and**, **or**, and **xor** bi-decompositions of  $h$  are complete,  $h$  is not bi-decomposable. Thus,  $h$  is relaxed to  $h'$  by minimally introducing don't cares in the on-set of  $h$  using a universal quantification of  $h$ ,  $\forall_x h$ , with respect to one variable  $x$ . Since  $\forall_b h$ ,  $\forall_c h$ , and  $\forall_d h$ , cover the same number of minterms in the on-set of  $h$ , we choose  $\forall_b h = \bar{c}\bar{d}$  as the don't care set of  $h'$ . The bi-decomposition for the relaxed function,  $h'$ , has a disjoint **xor** bi-decomposition (see Figure 7.4(iii)). Thus,  $f = \bar{a}c \oplus (\bar{c}\bar{d} + (b \oplus c \oplus d))$  (see Figure 7.4(iv)).

We will start by illustrating our BEG-based bi-decomposition technique on the 4-input logic function shown in Figure 7.4(i). First, we construct the BEGs of  $f$  for **and**, **or**, and **xor** bi-decompositions. Since the BEG for **and** and **or** bi-decompositions are complete graphs, there are no **and** or **or** bi-decompositions for  $f$ . The BEG for the **xor** bi-decomposition of  $f$  is not a complete graph and has the set  $c$  as a vertex cut of the BEG. Hence, there is an overlapping variable partition  $(\{a, c\}, \{b, c, d\})$  for an **xor** bi-decomposition of  $f$ , i.e.,  $f = g(a, c) \oplus h(b, c, d)$ . Using algorithm 4, it is

determined that the variable partition is feasible and the decomposed functions  $g$  and  $h$  are also obtained. Since  $g$  is the simple two input function  $\bar{a}c$ , Figure 7.4 does not show the steps for the obtaining the decomposition of  $g$ .

The bi-decomposition of  $h$  is the next recursive step and is shown in Figure 7.4(ii). Since the BEG for the **and**, **or**, and **xor** bi-decompositions of  $h$  are complete,  $h$  is not bi-decomposable. Thus,  $h$  is relaxed to  $h'$  by minimally introducing don't cares in the on-set of  $h$  using a universal quantification of  $h$ ,  $\forall_x h$ , with respect to one variable  $x$ . Since  $\forall_b h$ ,  $\forall_c h$ , and  $\forall_d h$ , cover the same number of minterms in the on-set of  $h$ , we choose  $\forall_b h = \bar{c}\bar{d}$  as the don't care set of  $h'$ . The bi-decomposition for the relaxed function,  $h'$ , has a disjoint **xor** bi-decomposition (see Figure 7.4(iii)). Thus,  $f = \bar{a}c \oplus (\bar{c}\bar{d} + (b \oplus c \oplus d))$  (see Figure 7.4(iv)).

Table 7.1 : Comparison of the proposed technique with the best algorithms in FBDD, SIS, ABC, and the industrial tool<sup>†</sup>

Name	FBDD [34]			SIS [80]			ABC [37]			Industry tool			BEG		
	Levels	Delay	Power	Levels	Delay	Power	Levels	Delay	Power	Levels	Delay	Power	Levels	Delay	Power
cordic	12	3.69	1.47	9	3.64	1.69	10	3.23	2.39	9	3.34	2.44	7	3.35	1.64
dalu	48	7.23	17.8	20	7.17	22.4	31	6.09	25.5	14	7.00	13.9	10	5.49	33.3
t481	6	3.48	0.96	13	3.67	4.2	14	5.59	13.9	11	4.32	7.7	6	3.48	0.96
C432	33	9.63	11.9	22	8.62	15.9	23	9.77	7.9	20	9.84	9.7	11	5.23	36.3
alu2	43	9.07	14.6	22	8.64	13.1	32	8.59	15.7	21	8.69	13.3	10	4.18	13.9
alu4	21	6.25	158.4	12	6.03	55.7	12	6.24	55.9	13	6.11	56.9	10	5.03	87.1
apex4	24	6.2	134.6	12	6.06	55.5	12	6.37	49	13	5.98	60.4	10	5.03	87
term1	18	4.08	6.1	11	3.55	8.2	12	3.69	5.8	10	3.96	7.1	9	3.72	3.2
frg1	13	2.95	1.6	11	3.56	4.1	12	3.71	1.8	10	3.41	4.2	8	2.68	1.7
i7	6	2.23	20.8	5	2.23	42.2	5	2.24	18.3	6	2.23	23.1	5	2.07	20.5
i8	17	5.66	22.7	11	5.51	28.6	11	5.23	39.9	10	5.91	22.6	9	4.7	37.8
too_large	52	6.71	34.1	12	5.01	15.3	24	4.95	13.7	13	5.44	13.3	10	3.91	7.1
lsu_stb_rwctl	17	5.54	31.6	10	5.64	31.6	15	5.68	31.1	11	5.81	29.9	11	5.55	36.5
sparc_tlu_intctl	–	–	–	8	3.44	12.7	11	3.5	11.6	8	3.29	12.1	7	3.01	12.7
sasc	–	–	–	8	3.33	30	8	2.7	28.8	7	3.33	27.3	6	3.11	22.8
spi	–	–	–	–	–	–	28	13.44	78	26	13.06	87.4	18	9.16	193.7
<b>Relative average</b>	2.44	1.25	0.96	1.5	1.23	0.88	1.81	1.18	0.83	1.43	1.25	0.78	1	1	1

<sup>†</sup> FBDD: default; SIS: delay, rugged, algebraic, and speed\_up;  
ABC: resyn2rs; Industry-standard synthesizer: -map-effort high -area-effort high and set\_max\_delay 0



Table 7.2 : Comparison of the proposed technique with the best algorithms in FBDD, SIS, ABC, and the industrial tool

Name	PI/POs	FBDD [34]		SIS [80]		ABC [37]		Industry tool		BEG	
		Gates <sup>†</sup>	Area <sup>*</sup>	Gates <sup>†</sup>	Area <sup>*</sup>	Gates <sup>†</sup>	Area <sup>*</sup>	Gates <sup>†</sup>	Area <sup>*</sup>	Gates <sup>†</sup>	Area <sup>*</sup>
cordic	23/2	62	99	88	98	53	151	65	146	32	105
dalv	75/16	1201	1190	1604	1426	1046	1515	707	1022	862	1911
t481	16/1	25	72	1227	202	742	936	159	510	25	72
C432	36/7	235	492	273	399	136	437	218	443	826	1487
alu2	10/6	484	716	482	614	349	748	357	608	400	778
alu4	9/19	4771	5619	2194	4455	2007	4372	1610	4676	2356	5218
apex4	9/19	4540	5286	2194	4653	2003	3681	1589	4280	2349	5218
term1	34/10	262	338	323	440	147	346	186	443	94	270
frg1	28/3	51	101	110	226	84	176	130	227	51	101
i7	199/67	510	1365	650	1431	568	1234	510	1629	500	1692
i8	133/81	979	1291	1621	1435	892	1635	1212	1180	1101	1786
too_large	38/3	1585	1639	614	894	392	748	434	695	172	434
lsu_stb_rwctl	250/205	490	1029	562	1095	482	998	587	969	755	1272
sparc_tlu_intctl	82/80	–	–	227	739	174	682	241	711	221	794
sasc	250/132	–	–	776	1451	547	1453	683	1374	536	1072
spi	505/227	–	–	–	–	3158	3888	3083	4131	4623	8274
<b>Relative average</b>	–	1.95	1.1	4.82	1.2	2.87	1.74	1.54	1.4	1	1

<sup>†</sup>The number of gates is reported after decomposition as the number of nodes in the AIG.

<sup>\*</sup>Area is reported after the decomposed AIG is mapped using the industrial tool in all cases.

Our bi-decomposition technique is implemented within ABC [37]. Given a circuit, each output is represented by BDDs using the CUDD package [85]. Then, each output is recursively decomposed into smaller sub-functions using BEGs. BEGs are stored and manipulated using the `igraph` library [86]. The variable partition for the bi-decomposition of a logic function is obtained from the minimum vertex cuts of the BEG. Our implementation obtains the minimum vertex cut of the BEG of an undirected graph with  $n$  vertices by converting the undirected graph into a directed flow graph with  $2n$  vertices. The minimum edge cut of the directed flow graph, obtained using the algorithm described in [87], is then used to obtain the minimum vertex cut of the undirected graph.

**Computational complexity:** The bulk of the computational time for a single level of bi-decomposition lies in finding a feasible variable partition for the `and`, `or`, and `xor` bi-decompositions. For variable partitioning using BEGs, a major portion of computational time is used for computing  $O(n^2)$  ( $n$  is the number of input variables) blocking conditions for building the BEGs for `and`, `or`, and `xor` bi-decompositions. For instance, the largest function considered that we have considered has 149 variables and the CPU time required for constructing the BEG for the `and`, `or`, and `xor` bi-decompositions of this function is 218 secs and the CPU time required to obtain the variable partitions from these BEG is 67 secs. Note that our simulations use a single processor for computation, but variable partitioning algorithm using BEGs is easily parallelizable because the blocking condition between different variable pairs

can be computed in parallel without any communication overhead. For the benchmark circuits reported in Table 7.1, the runtimes for all synthesis tools is in the order of a few minutes.

The decomposition for some benchmark circuits, e.g., C880 from the ISCAS benchmark suite, has a runtime in the order of hours using our implementation of bi-decomposition based on BEGs. The large runtime for these benchmarks is not due to the computational complexity of the variable partitioning algorithm based on BEGs. Instead, the reason for the large runtime is that these benchmark circuits have large non bi-decomposable Boolean functions and the relaxation heuristic that we use to decompose non bi-decomposable functions is not effective for these benchmark circuits. Hence, decomposition for these circuits is not only time-consuming, but also the final decomposed circuit has a large delay, area, and power. Obtaining optimal variable partitions non bi-decomposable functions is an open problem and is not addressed in this thesis.

**Redundancy removal:** Our implementation also performs area recovery using a function-based redundancy removal technique. Since bi-decomposition is performed in a depth-first recursive manner, bi-decomposed functions are cached in a hash table. If the function is encountered again in the same circuit, then the cached bi-decomposition is reused.

We compare our BEG-based bi-decomposition technique to state-of-the-art academic tools — FBDD [34], SIS [80], and ABC [37] — and an industry standard syn-

thesizer. Sixteen circuits from the MCNC, ISCAS, and IWLS benchmark suites and the OpenSPARC T1 processor are optimized using these synthesis tools on a 64-bit 2.4 GHz Opteron-based system. Each benchmark circuit is optimized with each tool to minimize the delay of the decomposed circuit. The decomposed circuit is mapped to the `lsi_10k` gate library that consists of 89 gates with the industry-standard tool.

The first column in table 7.1 is the name of the circuit. Subsequent columns report the number of levels of logic in the **and-invert** graph (AIG [37]), the mapped delay, and the dynamic power consumption at 1GHz for the results obtained with each optimization tool. For each benchmark circuit, the *best* results with the lowest mapped delay are reported in the table. For the BDD-based decomposition tool (FBDD), default synthesis options were used. Within SIS, the scripts `delay`, `rugged`, `algebraic`, and `speed_up` were used. Within ABC, script `resyn2rs` was used. Within the industry-standard synthesizer, each design was compiled with the options `-map-effort high` and `-area-effort high` and the design constraint `max_delay` was set to 0. The last row in the table compares the average results across the tools, normalized to the results of the industry-standard tool. On average, our technique shows a 60%, 34%, 45% and 30% reduction in the number of logic levels in the optimized circuit over FBDD, SIS, ABC, and the industry-standard synthesizer, respectively. When mapped delays are evaluated, our technique achieves an average reduction of 20%, 19%, 16% and 20% over the best results of FBDD, SIS, ABC, and the industry-standard synthesizer, respectively. For our technique, the trade-off for a 20% improvement in mapped

delay over the industry-standard synthesizer is a 28% increase in the dynamic power consumption.

Table 7.2 presents results to compare the number of gates in the AIG and the mapped area of our technique with state-of-the-art logic optimization tools. The first two columns in table 7.2 give the circuit information. Subsequent columns report the number of gates and the mapped area of the logic circuit for each tool.

**Area-delay trade-off:** As discussed in Section 7.2.3, variable partitions with smaller  $\Sigma$  typically yield bi-decompositions with lower area and power, whereas variable partitions with smaller  $\Delta$  typically yield circuits with lower delay. We have observed that for most circuits the best delay, area, and power is achieved by selecting the variable partition with the smallest  $\Sigma$ . However, some circuits, e.g., `dal`, `sasc`, and `alu2`, exhibit an area versus delay trade-off where reductions in the delay of the decomposed circuit can be achieved when variable partitions with lower  $\Delta$  are chosen.

This chapter described a new approach for obtaining variable partitions for the bi-decomposition of logic functions. Disjoint and overlapping variable partitions for `and`, `or`, and `xor` bi-decompositions of a logic function were obtained from the vertex cuts of an undirected graph called the *blocking edge graph*. Using this technique, an average reduction in delay of 20% was achieved for an average power overhead of 28% over the best results of an industry-standard synthesis tool across 16 benchmark circuits.

## Chapter 8

### Conclusions and future research

This thesis proposed a general theory of approximation for Boolean specifications. For a given specification, represented as a logic circuit, this thesis also proposed efficient algorithms for synthesis of an approximate logic circuit. The synthesis algorithms were based on a divide-and-conquer approach in which a given logic circuit is approximated by combining the approximations of small cluster of gates within the given logic circuit. The synthesis algorithms were designed to minimize the hardware overhead (area, power, and delay) of the approximate logic circuit while being able to target a specified input sub-space for approximation.

This thesis also demonstrated the application of approximate logic circuits to improve reliability of designs. By targeting an input sub-space for which the outputs of a logic circuit are most vulnerable to errors, approximate logic circuits were demonstrated to improve reliability of a logic circuit to errors arising due to a wide range of failure mechanisms. Specifically, this thesis demonstrated that approximate logic circuits can provide hardware support for online error detection/masking of logical errors arising due to transient failures, e.g., single-event upsets, and timing errors arising due to dynamic variability. To further reduce combinational logic overhead for timing error masking, this thesis proposed new time-borrowing latch/flip-flop designs

to mask timing errors at outputs with large timing slack.

Finally, this thesis demonstrated that logic decomposition based on approximate logic circuits can be used to push the envelope on frequency of operation for high-performance applications. The insights into logic decomposition obtained from approximate logic circuits facilitated the development of a new algorithm to obtain optimal variable partitions for bi-decomposition of Boolean functions. The variable partitioning algorithm provided further improvements in performance, especially for xor-dominant logic circuits.

This thesis has formulated a theoretical platform for circuit approximation and demonstrated application of approximate logic circuits to improve reliability and performance of designs. Several potential applications of approximate logic circuits that were not explored in this thesis include:

- This thesis demonstrated the application of approximate logic circuits to concurrent error detection and masking of logical errors arising due transient failures, e.g., single-event upsets. The same principle can be used to detect and mask logical errors arising due to intermittent failures, e.g., latent manufacturing defects. However, since modeling intermittent failures and evaluating their impact on logic circuits is a challenge, the application of approximate logic circuits to detect/mask logical errors arising due to intermittent failures has been left as an open problem.
- In the past, speculative computation has mainly been explored for regular de-

signs such as adders and branch-predictors. The theoretical formulation of an approximation and synthesis algorithms for approximate logic circuits, proposed in this thesis, broaden the scope of application for speculative computation techniques to arbitrary multi-level logic circuits.

Besides exploring new applications of approximate logic circuits, this thesis identifies several open problems in the domain of Boolean function decomposition and synthesis of approximate logic circuits.

- Chapter 7 proposed an algorithm for obtaining optimal variable partition for bi-decomposition of Boolean functions using blocking edge graphs. However, this algorithm can obtain variable partitions only for bi-decomposable functions. Obtaining variable partitions for decomposing non bi-decomposable functions has been left as an open problem.
  - The blocking condition proposed in Chapter 7 to obtain variable partitions during a bi-decomposition provides insights into exploring metrics for quantifying the contribution of a minterm towards the complexity of a Boolean function. For instance, minterms that are contained in `and`, `or`, and `xor` squares increase the complexity of Boolean function because they hinder the decomposability of a Boolean function. Such insights can be used to explore better approximations of Boolean functions that are easily decomposable into approximate logic circuits with a small delay, area, and power.
-



- Advances in bi-decomposition of Boolean functions proposed in Chapter 7 also opens up new directions for improving synthesis of approximate logic circuits. Bi-decomposition algorithms bridge the gap between Boolean functions and logic circuits, thus enabling approximate logic circuits to be synthesized directly from approximate functions. In other words, approximate functions can be explored in a more effective manner using representations like binary decision diagrams. The approximate functions can then be decomposed and synthesized into an approximate logic circuit using the bi-decomposition algorithm described in Chapter 7.

## Bibliography

- [1] M. Orshansky, S. Nassif, and D. Boning, *Design for Manufacturability and Statistical Design: A Constructive Approach*. Springer, 2008.
  - [2] T. Nigam, A. Kerber, and P. Peumans, “Accurate model for time-dependent dielectric breakdown of high-k metal gate stacks,” in *Proc. Intl. Reliability Physics Symposium*, pp. 523–530, 2009.
  - [3] A. Agarwal, D. Blaauw, and V. Zoltov, “Statistical timing analysis for intra-die process variations with spatial correlations,” in *Proc. Intl. Conference Computer-aided Design*, pp. 900–907, 2003.
  - [4] D. Pan and M. Cho, “Synergistic physical synthesis for manufacturability and variability in 45nm designs and beyond,” in *Proc. Design Automation Conference, Asia and South Pacific*, pp. 220–225, 2008.
  - [5] J. von Neumann, “Probabilistic logics and the synthesis of reliable organisms from unreliable components,” in *Automata Studies* (C. E. Shannon and J. McCarthy, eds.), pp. 43–98, Princeton University Press, 1956.
  - [6] S. Kundu and S. M. Reddy, “On symmetric error correcting and all unidirectional error detecting codes,” *IEEE Trans. Computers*, vol. 39, pp. 752–761, Jun. 1990.
-

- [7] N. A. Touba and E. J. McCluskey, "Logic synthesis of multilevel circuits with concurrent error detection," *IEEE Trans. Computer-aided Design*, vol. 16, pp. 783–789, Jul. 1997.
- [8] P. Nigh and A. Gattiker, "Test method evaluation experiments & data," in *Proc. Intl. Test Conference*, pp. 454–463, 2000.
- [9] K. Mohanram and N. A. Touba, "Cost-effective approach for reducing soft error failure rate in logic circuits," in *Proc. Intl. Test Conference*, pp. 893–901, 2003.
- [10] K. Mohanram and N. A. Touba, "Partial error masking to reduce soft error failure rate in logic circuits," in *Proc. Defect and Fault Tolerance Symposium*, pp. 433–440, 2003.
- [11] S. Krishnaswamy, S. M. Plaza, I. L. Markov, and J. P. Hayes, "Enhancing design robustness with reliability-aware resynthesis and logic simulation," in *Proc. Intl. Conference Computer-aided Design*, pp. 149–154, 2007.
- [12] S. Almkhaizim, Y. Makris, Y. s. Yang, and A. Veneris, "On the minimization of potential transient errors and SER in logic circuits using SPFD," in *Proc. Intl. On-line Testing Symposium*, pp. 123–128, 2008.
- [13] Y. Tsiatouhas, S. Matakias, A. Arapoyanni, and T. Haniotakis, "A sense amplifier based circuit for concurrent detection of soft and timing errors in CMOS ICs," in *Proc. Intl. On-line Testing Symposium*, pp. 12–16, 2003.

- [14] S. Mitra *et al.*, “Robust system design with built-in soft error resilience,” *IEEE Computer*, vol. 38, pp. 43–52, Feb. 2005.
  - [15] P. Franco and E. J. McCluskey, “On-line delay testing of digital circuits,” in *Proc. VLSI Test Symposium*, pp. 167–173, 1994.
  - [16] M. Favalli and C. Metra, “Sensing circuit for on-line detection of delay faults,” *IEEE Trans. VLSI Systems*, vol. 4, pp. 130–133, 1996.
  - [17] M. Agarwal, B. C. Paul, M. Zhang, and S. Mitra, “Circuit failure prediction and its application to transistor aging,” in *Proc. VLSI Test Symposium*, vol. 32, pp. 277–286, 2007.
  - [18] M. Nicolaidis, “Time redundancy based soft error tolerance to rescue nanometer technologies,” in *Proc. VLSI Test Symposium*, pp. 86–94, 1999.
  - [19] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, “Razor: A low-power pipeline based on circuit-level timing speculation,” in *Proc. Intl. Symposium on Microarchitecture*, pp. 7–18, 2003.
  - [20] T. Sato and Y. Kunitake, “A simple flip-flop circuit for typical-case designs for DFM,” in *Proc. Intl. Symposium on Quality Electronic Design*, pp. 539–544, 2007.
  - [21] K. A. Bowman, J. W. Tschanz, N. S. Kim, J. C. Lee, C. B. Wilkerson, S.-L. Lu,
-

- T. Karnik, and V. K. De, “Energy-efficient and metastability-immune timing-error detection and recovery circuits for dynamic variation tolerance,” in *Proc. Intl. Conference on Integrated Circuit Design and Technology*, pp. 155–158, 2008.
- [22] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, “Parameter variations and impact on circuits and microarchitecture,” in *Proc. Design Automation Conference*, pp. 338–342, 2003.
- [23] S. Mitra and E. J. McCluskey, “Design of redundant systems protected against common-mode failures,” in *Proc. VLSI Test Symposium*, pp. 190–195, 1997.
- [24] W. Kunz and D. Pradhan, “Recursive learning: A new implication technique for efficient solutions to CAD problems — Test, verification, and optimization,” in *IEEE Trans. Computer-aided Design*, vol. 13, pp. 1143–1158, 1994.
- [25] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh, and M. Papaefthymiou, “Precomputation-based sequential logic optimization for low power,” in *Proc. Intl. Conference Computer-aided Design*, pp. 74–81, 1994.
- [26] A. Saldanha, H. Harkness, P. C. McGeer, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, “Performance optimization using exact sensitization,” in *Proc. Design Automation Conference*, pp. 425–429, 1994.
- [27] J. Monteiro, J. Rinderknecht, S. Devadas, and A. Ghosh, “Optimization of combinational and sequential logic circuits for low power using precomputation,” in *Proc. Conf. Advanced Research in VLSI*, pp. 430–444, 1995.
-

- [28] J. Monteiro, S. Devadas, and A. Ghosh, "Sequential logic optimization for low power using input-disabling precomputation architectures," *IEEE Trans. Computer-aided Design*, vol. 17, pp. 279–284, 1998.
  - [29] T. Xia, J. Feng, Z. Chen, and L. Ji, "A new precomputation architecture of sequential logic circuits for low power," pp. 2071–2074, 2004.
  - [30] S.-L. Lu, "Speeding up processing with approximation circuits," *Computer*, vol. 37, pp. 67–73, 2004.
  - [31] A. K. Verma, P. Brisk, and P. Ienne, "Variable latency speculative addition: a new paradigm for arithmetic circuit design," in *Proc. Design Automation and Test in Europe*, pp. 1250–1255, 2008.
  - [32] G. Hachtel and F. Somenzi, *Logic synthesis and verification*. Kluwer Academic Publishers, 2000.
  - [33] C. Yang and M. Ciesielski, "BDS: A BDD-based logic optimization system," *IEEE Trans. Computer-aided Design*, vol. 21, pp. 866–876, 2000.
  - [34] D. Wu and J. Zhu, "FBDD: A folded logic synthesis system," in *Intl. Conference on ASIC*, pp. 746–751, 2005.
  - [35] G. de Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.
-

- [36] T. Sasao, ed., *Logic synthesis and optimization*. Kluwer Academic Publishers, Boston, MA, 1993.
  - [37] “ABC Logic synthesis tool.” Please visit the URL <http://www.eecs.berkeley.edu/~alanmi/abc/> for further details.
  - [38] D. K. Pradhan, ed., *Fault-tolerant computer system design*. Prentice-Hall, Inc., NJ, USA, 1996.
  - [39] M. Gössel and S. Graf, *Error detection circuits*. McGraw-Hill Book Company, London, UK, 1993.
  - [40] N. K. Jha and S. Wang, “Design and synthesis of self-checking VLSI circuits,” *IEEE Trans. Computer-aided Design*, vol. 2, pp. 878–887, 1993.
  - [41] V. V. Saposhnikov, V. V. Saposhnikov, A. Dmitriev, and M. Goessel, “Self-dual duplication for error detection,” in *Proc. Asian Test Symposium*, pp. 296–300, 1998.
  - [42] D. Das and N. Toubia, “Synthesis of circuits with low cost concurrent error detection based on Bose-Lin codes,” in *Journal of Electronic Testing: Theory and Applications*, vol. 15, pp. 145–155, 1999.
  - [43] S. Almukhaizim, P. Drineas, and Y. Makris, “Entropy-driven parity tree selection for low-cost concurrent error detection,” *IEEE Trans. Computer-aided Design*, vol. 25, pp. 1547–1554, 2006.
-

- [44] D. K. Pradhan, ed., *Fault-tolerant computing: Theory and techniques*, vol. 1. Prentice-Hall, NJ, USA, 1986.
  - [45] S. Krishnaswamy, G. F. Viamontes, I. L. Markov, and J. P. Hayes, “Accurate reliability evaluation and enhancement via probabilistic transfer matrices,” in *Proc. Design Automation and Test in Europe*, pp. 282–287, 2005.
  - [46] T. Rejimon and S. Bhanja, “Scalable probabilistic computing models using Bayesian networks,” in *Proc. Intl. Midwest Symposium on Circuits and Systems*, pp. 712–715, 2005.
  - [47] M. R. Choudhury and K. Mohanram, “Accurate and scalable reliability analysis of logic circuits,” in *Proc. Design Automation and Test in Europe*, pp. 1454–1459, 2007.
  - [48] J. Blome, S. Feng, S. Gupta, and S. Mahlke, “Online timing analysis for wearout detection,” in *Workshop on architectural reliability*, 2006.
  - [49] K. A. Bowman, J. W. Tschanz, N. S. Kim, J. C. Lee, C. B. Wilkerson, S.-L. Lu, T. Karnik, and V. K. De, “Energy-efficient and metastability-immune timing-error detection and instruction-replay-based recovery circuits for dynamic-variation tolerance,” in *Proc. Intl. Solid-state Circuits Conference*, pp. 402–403, 2008.
  - [50] K. A. Bowman, J. W. Tschanz, C. B. Wilkerson, S.-L. Lu, T. Karnik, V. K. De,
-



and S. Borkar, “Circuit techniques for dynamic variation tolerance,” in *Proc. Design Automation Conference*, pp. 4–7, 2009.

- [51] Y.-S. Su, D.-C. Wang, S.-C. Chang, and M. Marek-Sadowska, “An efficient mechanism for performance optimization of variable-latency designs,” in *Proc. Design Automation Conference*, pp. 976–981, 2007.
  - [52] L. Benini, E. Macii, M. Poncino, and G. De Micheli, “Telescopic units: A new paradigm for performance optimization of VLSI designs,” *IEEE Trans. Computer-aided Design*, vol. 17, pp. 220–232, 1998.
  - [53] L. Benini, G. De Micheli, A. Liroy, E. Macii, G. Odasso, and M. Poncino, “Automatic synthesis of large telescopic units based on near-minimum timed super-setting,” *IEEE Trans. Computers*, vol. 48, pp. 769–779, 1999.
  - [54] J. C. Smolens, B. T. Gold, J. C. Hoe, B. Falsafi, and K. Mai, “Detecting emerging wearout faults,” in *Workshop on Silicon Errors in Logic - System Effects*, pp. 276–287, 2007.
  - [55] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, “The case for lifetime reliability-aware microprocessors,” in *Proc. Intl. Symposium on Computer Architecture*, pp. 276–287, 2004.
  - [56] J. Tschanz, N. S. Kim, S. Dighe, J. Howard, G. Ruhl, S. Vanga, S. Narendra, Y. Hoskote, H. Wilson, C. Lam, M. Shuman, C. Tokunaga, D. Somasekhar,
-

- S. Tang, D. Finan, T. Karnik, N. Borkar, N. Kurd, and V. De, "Adaptive frequency and biasing techniques for tolerance to dynamic temperature-voltage variations and aging," in *Proc. Intl. Solid-state Circuits Conference*, pp. 292–493,604, 2007.
- [57] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoCs," in *Proc. Design Automation Conference*, pp. 7–12, 2006.
- [58] A. B. T. Hopkins and K. D. McDonald-Maier, "Debug support for complex systems on-chip: a review," in *Proc. Computers and Digital Techniques*, pp. 197–207, 2006.
- [59] J.-S. Yang and N. A. Touba, "Expanding trace buffer observation window for in-system silicon debug through selective capture," in *Proc. VLSI Test Symposium*, pp. 345–351, 2008.
- [60] C. Metra, M. Favalli, and B. Ricco, "On-line detection of logic errors due to crosstalk, delay, and transient faults," in *Proc. Intl. Test Conference*, pp. 524–533, 1998.
- [61] A. Paschalis, D. Gizopoulos, and N. Gaitanis, "Concurrent delay testing in totally self-checking systems," in *Journal of Electronic Testing: Theory and Applications*, vol. 12, pp. 55–61, 1998.
-

- [62] B. C. Paul, K. Kang, H. Kufluoglu, M. A. Alam, and K. Roy, "Impact of NBTI on the temporal performance degradation of digital circuits," *IEEE Electron Device Letters*, vol. 26, pp. 560–562, 2005.
  - [63] M. R. Choudhury and K. Mohanram, "Masking timing errors on speed-paths in logic circuits," in *Proc. Design Automation and Test in Europe*, pp. 87–92, 2009.
  - [64] M. Kurimoto, H. Suzuki, R. Akiyama, T. Yamanaka, H. Ohkuma, H. Takata, and H. Shinohara, "Phase-adjustable error detection flip-flops with 2-stage hold driven optimization and slack based grouping scheme for dynamic voltage scaling," in *Proc. Design Automation Conference*, pp. 884–889, 2008.
  - [65] K. Hirose, Y. Manzawa, M. Goshima, and S. Sakai, "Delay-compensation flip-flop with *in-situ* error monitoring for low-power and timing-error-tolerant circuit design," *Japanese Journal of Applied Physics*, vol. 47, pp. 2779–2787, 2008.
  - [66] M. Ghasemazar, B. Amelifard, and M. Pedram, "A mathematical solution to power optimal pipeline design by utilizing soft-edge flip-flops," in *Proc. Intl. Symposium on Low Power Electronics and Design*, pp. 33–38, 2008.
  - [67] R. L. Ashenhurst, "The decomposition of switching functions," *Computation Lab, Harvard University*, vol. 29, pp. 74–116, 1959.
  - [68] K. J. Singh, A. R. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Timing optimization of combinational logic," in *Proc. Intl. Conference Computer-aided Design*, pp. 282–285, 1988.
-

- [69] C. L. Berman, D. J. Hathaway, A. S. LaPaugh, and L. H. Trevillyan, "Efficient techniques for timing correction," in *Proc. Intl. Symposium on Circuits and Systems*, pp. 415–419, 1990.
  - [70] P. C. McGeer, R. K. Brayton, A. L. Sangiovanni-Vincentelli, and S. Sahni, "Performance enhancement through the generalized bypass transform," in *Proc. Intl. Conference Computer-aided Design*, pp. 184–187, 1991.
  - [71] K. Keutzer, S. Malik, and A. Saldanha, "Is redundancy necessary to reduce delay?," *IEEE Trans. Computer-aided Design*, vol. 10, no. 4, pp. 427–435, 1991.
  - [72] H. J. Touati, H. Savoj, and R. K. Brayton, "Delay optimization of combinational logic circuits by clustering and partial collapsing," in *Proc. Intl. Conference Computer-aided Design*, pp. 188–191, 1991.
  - [73] K.-C. Chen and S. Muroga, "Timing optimization for multi-level combinational networks," in *Proc. Design Automation Conference*, pp. 339–344, 1991.
  - [74] Y.-T. Lai, K.-R. R. Pan, and M. Pedram, "OBDD-based function decomposition: Algorithms and implementation," *IEEE Trans. Computer-aided Design*, vol. 15, no. 8, pp. 977–990, 1996.
  - [75] B. Becker, R. Drechsler, and M. Theobald, "On the expressive power of OKFDDs," *Formal Methods in System Design*, vol. 11, no. 1, pp. 5–21, 1997.
  - [76] M. Fujita, Y. Matsunaga, and M. Ciesielski, "Multi-level logic optimization," in
-

*Logic synthesis and verification* (S. Hassoun, T. Sasao, and R. K. Brayton, eds.), ch. 2, Kluwer Academic Publishers, Boston, MA, 2002.

- [77] T. Liu and S.-L. Lu, “Performance improvement with circuit-level speculation,” in *Proc. Intl. Symposium on Microarchitecture*, pp. 348–355, 2000.
- [78] R. Ladner and M. Fischer, “Parallel prefix computation,” *Journal of Association for Computing Machinery*, vol. 27, pp. 831–838, 1980.
- [79] S. Lakshmivarahan and S. K. Dhall, *Parallel Computing Using the Prefix Problem*. Oxford University Press, 1994.
- [80] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, “SIS: A system for sequential circuit synthesis,” Tech. Rep. UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.
- [81] H. Ling, “High speed binary parallel adder,” *IEEE Trans. Computers*, vol. 15, no. 5, pp. 799–802, 1966.
- [82] A. Mishchenko, B. Steinbach, and M. Perkowski, “An algorithm for bi-decomposition of logic functions,” in *Proc. Design Automation Conference*, pp. 103–108, 2001.
- [83] H.-P. Lin, J.-R. Jiang, and R.-R. Lee, “To SAT or not to SAT: Ashenhurst

decomposition in a large scale,” in *Proc. Intl. Conference Computer-aided Design*, pp. 32–37, 2008.

- [84] B. Steinbach, “Synthesis of multi-level circuits using exor-gates,” in *Proc. Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, pp. 161–168, 1995.
- [85] *CUDD: Colorado University Decision Diagram Package*. Please visit the URL <http://vlsi.colorado.edu/~fabio/CUDD/> for further details.
- [86] *The igraph library for complex network research*. Please visit the URL <http://igraph.sourceforge.net/> for further details.
- [87] Y. Boykov and V. Kolmogorov, “An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision,” *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 26, pp. 1124–1137, 2004.